(2)

RADC-TR-89-259, Vol II (of twelve)
Interim Report
October 1989

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT - 1988 Discussing, Using, and Recognizing Plans (NLP)

Syracuse University

Stuart C. Shapiro and Beverly Woolf

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-259, Vol II (of twelve) has been reviewed and is approved for publication.

APPROVED: Sharon M. Walter

SHARON M. WALTER
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-259, Vol II (of twelve)
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC)	6b. OFFICE SYMBOL (If applicable)
	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) Science & Technology Center, Rm 2-296 111 College Place, Syracuse University Syracuse NY 13244-4100	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL (If applicable) COES
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
62702F	5581	27	13

11. TITLE (Include Security Classification)
NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT - 1988 Discussing, Using, and Recognizing Plans (NLP)

12. PERSONAL AUTHOR(S)
Stuart C. Shapiro, Beverly Woolf

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM Jan 88 TO Dec 88	October 1989	100

16. SUPPLEMENTARY NOTATION This effort was funded partially by the Laboratory Directors' Fund. This effort was performed as a subcontract by the State University of New York at Buffalo and the University of Massachusetts at Amherst to Syracuse University (Continued on reverse)

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Artificial Intelligence, User Interfaces, Hypothetical Reasoning, Planning, Natural Language, Plan Recognition

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Reseasrch. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the fourth year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the recognition of plans expressed in natural language, followed by their discussion and use.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Sharon M. Walter	22b. TELEPHONE (Include Area Code) (315) 330-3577	22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Item 10. SOURCE OF FUNDING NUMBERS (Continued)

Program Element Number	Project Number	Task Number	Work Unit Number
62702F	5581	27	23
61102F	2304	J5	01
61102F	2304	J5	15
33126F	2155	02	10
61101F	LDFP	27	01

Item 16. SUPPLEMENTARY NOTATION (Continued)

Office of Sponsored Programs.

2 DISCUSSING, USING, AND RECOGNIZING PLANS

Report submitted by:

Dr. Stuart C. Shapiro, Principal Investigator
Dr. Beverly Woolf, Principal Investigator
Deepak Kumar, Graduate Research Assistant
Syed S. Ali, Graduate Research Assistant
Penelope Sibun, Graduate Research Assistant
David Forster, Graduate Research Assistant
Scott Anderson, Graduate Research Assistant

Department of Computer Science
SUNY at Buffalo
226 Bell Hall
Buffalo, NY 14260

Computer and Information Science
Graduate Research Center
University of Massachusetts
Amherst, MA 01003

TABLE OF CONTENTS

2 DISCUSSING, USING AND RECOGNIZING PLANS

2.1 INTRODUCTION

This project, also known as the Natural Language Planning project, is a joint project of a research group at SUNY at Buffalo (UB), led by Dr. Stuart C. Shapiro, and a research group at the University of Massachusetts at Amherst led by Dr. Beverly Woolf. The project is devoted to the investigation of a knowledge representation design compatible with the intensional knowledge representation theory previously developed by Dr. Shapiro and his co-workers and capable of providing a natural language interacting system with the ability to discuss, use, and recognize plans. The UB group is responsible for: the development, improvement, and maintenance of the knowledge representation, reasoning, and natural language processing software to be used in the project; for developing a representation of plans and associated concepts; and for developing basic techniques for discussing plans in English, and for plan recognition. The Massachusetts group is responsible for analyzing the chosen domains, mainly the domain of tutoring interactions, for testing the developments of the Buffalo group by trying to apply them to these domains, and for suggesting changes to the representation of plans. With the support of the NAIC and of Texas Instruments, both groups are using TI Explorers to do their work.

2.1.1 Objectives

The objectives of this project are to:

1. design a representation for plans and rules for reasoning about plans within an established knowledge representation/reasoning (KRR) system; enhance the KRR system so that it can act according to such plans;

2. write a grammar to direct an established natural language processing (NLP) system to analyze English sentences about plans and represent the semantic/conceptual content of the sentences in the representation designed for objective (1); the resulting NLP system should be able to:

   (a) accept sentences describing plans

   (b) add the plans to its "plan library"

   (c) answer questions about the plans in its plan library

   (d) accept sentences describing the actions of others, and

   (e) recognize when those actions constitute the carrying out of a plan in its plan library.

The KRR system to be used is SNePS[18], and the NLP system to be modified for this purpose is CASSIE[22]. The UB group is responsible for enhancing SNePS/CASSIE according to the objectives listed above, using the Blocksworld as an initial development/testing domain. The U. Mass. group is responsible for testing the enhanced system in the specific domains of tutoring and space launch narratives.

2.1.2 Overview

This report describes in detail how the objectives outlined above are being met. We discuss the design, implementation, and use of representations for plans to model a cognitive agent whose behavior is driven by its beliefs, desires, and intentions. We will give the motivations underlying our representations for plans, goals, acts, actions, pre-conditions and post-conditions. These representations are designed to satisfy constraints posed by the issues involved in fulfilling the tasks mentioned above (natural language understanding, belief representation, planning and problem solving, plan recognition, and text generation).

SNePS, the knowledge representation/reasoning system being used for this project, and its associated Generalized Augmented Transition Network (GATN) grammar interpreter/compiler was implemented in Franz Lisp when this project started. Significant steps have been taken to reimplement the software, now called SNePS-2, in Common Lisp. This was crucial for cooperation between the UB and UMass groups, since both are using Common Lisp on TI Explorer Lisp Machines, but when 1988 began SNePS-2 was still missing SNIP, the inference package that carries out reasoning, and the GATN interpreter/compiler. With the help of the UMass group, the UB group has now completed the implementation of the SNePS-2 version of the GATN interpreter/compiler, and enough of SNIP-2 has been implemented (a superset of Horn Clause logic) to be useable by both groups. The current, though still incomplete, draft of the SNePS-2 User Manual is included as an appendix to this report.

Overview of the system

Our work is proceeding by implementing, experimenting with, and revising a system called SNACTor. SNACTor begins with an empty knowledge-base. In the role of informant, we interact with SNACTor using English sentences about the domain, instructing SNACTor about the various actions that it can perform, and how to solve problems in that domain. The input sentences are analyzed using a domain-specific grammar, the results of which are new beliefs in the knowledge-base. A natural language generation grammar takes the new beliefs and expresses them back in English to indicate SNACTor's understanding to the informant. Requests to perform some action are sent to an acting executive that may then generate and execute a plan to fulfill the request. The informant may also ask questions about plans and the way the system would solve various problems.

A Generalized ATN grammar [19] is used for analyzing input sentences and for generating English responses. SNACTor currently operates in a Blocksworld domain.

During FY87-88 the UMASS group developed a hybrid architecture for reasoning about plans in multi-sentential narratives. The architecture contains both a strategic planning framework, based on the SNePS-2/Cassie system described above, and a 'reactive-planning' framework. It uses a deep representation of general world knowledge, objects and human goals to guide comprehension of text and recognition of plans. We have also generated an architecture for disambiguating example- or case- based anaphora, such as in the word "situation" used to refer to a set of events or objects. Each of these research areas is discussed below.

2.2 MOTIVATIONS UNDERLYING OUR REPRESENTATIONS

Our goals are to design and implement representations for plans to model a rational cognitive agent whose behavior is driven by its beliefs, desires, and intentions. We now give the motivations underlying our representations of plans, goals, acts, actions, pre-conditions and post-conditions. As mentioned before these representations are designed to satisfy constraints posed by issues in natural language understanding, belief representation, planning and problem solving, plan recognition, and text generation. A preliminary version of this work appears as an extended abstract in [20]. Since then, there have been changes in the design of our representations and planning techniques used. A detailed account of the syntax, and semantics of our earlier representations can be found in [10].

2.2.1 Motivation for intensional representations of plans

Georgeff (1987) mentions the importance of "considering planning systems as rational agents that are endowed with the psychological attitudes of belief, desire, and intention" and the problem of using appropriate semantics that give an *intensional* account of these notions. SNePS is an intensional propositional semantic network system [22] that has been used for cognitive modeling, belief representation and reasoning, belief revision, and natural language understanding. A basic principle of SNePS is the Uniqueness Principle—that there be a one-to-one mapping between nodes of the semantic network and concepts (mental objects) about which information may be stored in the network. These concepts are not limited to objects in the real world, but may be various ways of thinking about a single real world object, such as The Morning Star *vs.* The Evening Star *vs.* Venus. They may be abstract objects like properties, propositions, Truth, Beauty, fictional objects, and impossible objects. They may include specific propositions as well as general propositions, and even rules. Any concept represented in the network may be the object of propositions represented in the network giving properties of, or beliefs about it. For example, propositions may be the objects of explicit belief (or disbelief) propositions. Rules are propositions with the additional property that SNIP, the SNePS Inference Package, [13, 21] can use them to drive reasoning to derive additional believed propositions from previous believed propositions.

Plans are also mental objects. We can discuss plans with each other, reason about them, formulate them, follow them, and recognize when others seem to be following them. An AI system, using SNePS as its belief structure, should also be able to do these things. Requiring that the system be able to use a single plan representation for all these tasks puts severe constraints on the design of the representation. For instance, understanding natural language dialogue involving plans requires building plan representations from natural language input. In natural language, the explication of plans generally takes the form of a sequence of rather simple rules (*e.g.,* "If you see John, tell him I'm looking for him," "To pick up a block, you must first clear it"). These rules can contain indefinite and definite

noun phrases and anaphoric references corresponding to typed plan variables. The full plan, including preconditions and effects of its component acts, must be constructed from such a sequence of rules.

Once constructed, a plan must be usable as a specification for the behavior of the agent, and must also be usable by the agent to understand other agents' actions. We are not treating plans as schedules of events for third parties (or multiple agents) [11].

2.3 INTENSIONAL REPRESENTATIONS

We now give an overview of our representations and the motivations that led to them. We use "goal," "plan," "act," and "action" in particular ways, and distinguish among them. A *goal* is a proposition in one of two roles—either the role within another proposition that some *plan* is a plan for achieving that goal (making it true in the then current world), or the role as the object of the *act* of achieving it.

2.3.1 Planning is different from inference

We view a *plan* as a structured individual mental concept, *i.e.*, it is not a proposition or rule that might have a belief status. A plan is a structure of *acts*. (Among which may be the achieving of some goal or goals.) The structuring syntax for plans is a special syntax, differing, in particular, from that used for structuring reasoning rules. This is important both for semantic clarity and to allow a system to be implemented that can both reason and act efficiently. For contrast, consider standard (non-concurrent) Prolog or some arbitrary production rule system. Such a system relies on a semantic ambiguity between the logical & and the procedural **and then**. For example,

(2.3.1) $$p(X) : -q(X), r(X)$$

either means "For any X, $p(X)$ *is true if* $q(X)$ and $r(X)$ are true" or it means "For any X, to do p on X, first do q on X and then do r on X." Guaranteeing the proper ordering of behavior in the procedural interpretation is only possible by giving up the freedom to reorder, for efficiency, the derivations of $q(X)$ and $r(X)$ in the logical interpretation. The example is made more striking by appending

(2.3.2) $$q(Y) : -s(Y), t(Y)$$

(2.3.3) $$r(Z) : -s(Z), u(Z)$$

Under the logical interpretation, it would be efficient for the system to try finding true instances of $s(X)$ only once, instead of once when rule 2.3.2 is being used and once when rule 2.3.3 is being used. This is the way SNIP has been implemented (see [13]). However, under the procedural interpretation, it is perfectly reasonable to perform $s(X)$ twice for a given X, so the behavior that optimizes logical reasoning destroys procedural rule following. The fact that SNIP is optimized in this way for reasoning, and so cannot use its reasoning rules as procedural rules, was what originally motivated this project to design a planning/acting component for SNePS.

Believing is a state of knowledge; acting is the process of changing one state into another. Reasoning rules pass a *truth* or a *belief* status from antecedent to consequent, whereas acting rules pass an *intention* status from earlier acts to later acts. A reasoning rule can be viewed as a rule specifying an act—that of believing some previously non-believed proposition, but

the believe action is already included in the semantics of the propositional connective, and, as pointed out above, there is no reason to believe a proposition more than once (unless it's disbelieved in the interim). The distinction between "believing and acting" in SNePS was first outlined in [14].

2.3.2 The distinction between "acts" and "actions"

Lifschitz (1987) attempts to give a semantics of STRIPS by viewing STRIPS as a form of logic and STRIPS operators as rules of inference in this logic. For us, an *act* is a structured individual mental concept of something that can be performed by various actors at various times. This is important for plan recognition—we must be able to recognize that another agent is performing the same act that, if we were performing it, we would be in the midst of carrying out one of a certain number of plans. By the Uniqueness Principle, a single act must be represented by a single SNePS node, even if there are several different structures representing propositions that several different actors performed that act at different times. This argues for a representation of propositions more like that of Almeida [1], rather than like more traditional case-based or frame-based representations. In what we are calling "more traditional representations", there is a structure representing the proposition with slots or arcs to the actor, the action, the object, *etc.* For example, to represent the proposition,

(s1) John walked to the store.

there would be four representational symbols, one for John, one for walking (or PTRANS-ing), one for the store, and one for the proposition itself, and the first three would be connected with the fourth in nearly similar ways at similar distances (measured by path length of arcs or slots). Almeida, however, took seriously that one could follow (s1) by

(s2) Mary did too.

and understand by that that John and Mary performed the same act—that of walking to the store. The representation for (s1) would have to introduce a fifth symbol, for walking to the store, which would be connected to the representation of the proposition at the same distance as the representation of John. Now, however, the symbols for walking and the store would be further from the symbol for the proposition. When (s2) is processed, the symbol representing the proposition that Mary walked to the store would be connected to the same symbol for walking to the store used for (s1). This symbol represents what we are calling an act, and using it in the representation of both propositions follows by the Uniqueness Principle from interpreting (s1) and (s2) as saying the John and Mary performed the same act. Moreover, if the network contains the representation of any plan that involves walking to the (same) store, that same act node would be used in the structure representing that plan. Thus, John and Mary are directly connected to a plan that they may be engaged in.

An *action* is that component of an act that is what is done to the object or objects. In (s1) and (s2), the action is walking. Achieving some goal is an act whose action is achieving,

and whose object is the particular proposition that is serving as the goal. Unfortunately for our remaining discussion, but consistently with what has gone before, one can only *perform* something that is an act (an action on an appropriate object), so instead of saying "performing an act whose action is x," we will say "performing the action x," and hope the reader will note the distinction between acts and actions.

Our representation of an act is a node with an ACTION arc to a node that represents the action, and OBJECT1, ..., OBJECTN arcs to the required objects of the action. Thus, the general syntax of an act is

Syn. 1: *act* ::= ACTION: *action*
OBJECT1: *object1*
...
OBJECTN: *objectN*

Sem. 1: *act* is a structured individual node representing the act whose action is *action* and *object1, ..., objectN* are the objects of *action*. For example, the SNePSUL (the SNePS command interpreter language) command for building a node representing the act of saying "FOO" is:

```
(build action say object1 FOO)
```

2.3.3 Primitive and Complex actions

Any behaving entity has a repertoire of *primitive actions* it is capable of performing. We will say that an act whose action is primitive is a *primitive act*. That an action is primitive is a belief held by SNACTor after we tell it. The belief is represented in the form of an assertion saying that the action is a member of the class of primitive actions. This is similar to the MEMBER-CLASS proposition used by CASSIE [22]. For example, the SNePSUL command for asserting that saying is a primitive action is

```
(assert member say class primitive)
```

Non-primitive acts, which we will term *complex*, can only be performed by decomposing them into a structure of primitive acts, the syntax of which is the same procedural syntax as used in plans. That some plan p is a plan for carrying out some complex act a, is a proposition we can assert to SNACTor using the following representation:

Syn. 2: *plan-act-proposition* ::= ACT: a
PLAN: p

Sem. 2: *plan-act-proposition* is a proposition node that represents that p is a plan for carrying out act a.

p is a structure of acts. The structuring syntax for plans is described in terms of control actions which are described later. That some plan p is a plan for achieving some goal g is also a proposition we can assert to SNACTor:

Syn. 3: *plan-goal-proposition* ::= GOAL: *g*
PLAN: *p*

Sem. 3: *plan-goal-proposition* is a proposition node that represents that *p* is a plan for achieving goal *g*.

g is expressed as a domain specific proposition. Examples of these are given in a later section (see 2.5).

When the time comes for the agent to perform a complex act, it must find a plan that decomposes it. Using the above representations SNACTor may be told such plans. SNACTor is also capable of doing classical planning in case it does not already know any decompositions for a complex act. This is discussed later.

2.3.4 Pre– and post–conditions

The remaining notions we must consider are preconditions and effects (postconditions). Whether we think of them as pre- and post-conditions of plans or of acts is irrelevant since plans are kinds of acts. A pre-(post-)condition is just a proposition that must be (will be) true or false before (after) an act is performed. But the proposition that a proposition *p* is false is itself a proposition, so we can say that a pre-(post-)condition is a proposition that must be (will be) *true* before (after) an act is performed. (We will rely on SNeBR, the SNePS Belief Revision System [3] to remove inconsistent beliefs after believing the effects of an act.) We have thus reduced the storage of pre- and post-conditions to two simple kinds of propositions:

Syn. 4: *precondition-proposition* ::= ACT: *a*
PRECONDITION: *p*

Sem. 4: *precondition-proposition* is a node that represents that the pre-condition of some act *a* is the proposition *p*.

For example,

```
(assert forall $block
        ant (build member *block class block)
        cq (build act (build action pickup object1 *block)
                  precondition (build property clear object *block)))
```

is the SNePSUL command to assert that before picking up any block first make sure that it is clear (*i.e.* there is nothing on top of it).

Syn. 5: *postcondition-proposition* ::= ACT: *a*
EFFECT: *p*

Sem. 5: *postcondition-proposition* is a node that represents that the post-condition of some act a is the proposition p.

For example,

```
(assert forall $block
        ant (build member *block class block)
        cq (build act (build action pickup object1 *block)
                  effect (build property holding object *block)))
```

is the SNePSUL command to assert that after picking up any block, it is being held.

Thus, effects and preconditions of an act are represented in the same way as other beliefs about other mental objects; we do not need a special data structure (or an operator formalism) for acts in which pre- and post-conditions are special fields. Such a representation also enables us to assert context-dependent effects of actions[27]; *i.e.*, the effects of doing some action are determined by the context in which the action is performed. For example,

```
(assert forall ($block $support)
        &ant ((build member *block class block)
              (build member *support class object)
              (build rel on arg1 *block arg2 *support))
        cq (build act (build action pickup object1 *block)
                  effect (build property clear object *support)))
```

asserts that if a block is on some support then after you pick up the block the support is clear. The scope of the context being referred to is the set of beliefs held by the system at the time the action is about to be performed. Using rules like these, context-dependency is guaranteed by ensuring that the effect is conditional on the antecedents being true before the act is performed. This is a more natural way of modeling actions and avoids the need for specifying multiple operators for doing the same action in different situations, which is a major criticism of earlier planners[6]. In section 2.6.4 we demonstrate this feature.

2.3.5 Types of actions

We discussed three kinds of acts: a *primitive* act is unstructured and is in the repertoire of the agent; a *complex* act is unstructured—to perform it, the agent must find a plan for it; a *plan* is a structured act—the structure determines how the agent performs the component acts.

The structure of a plan can determine how the agent performs the component acts, because the structure, itself, is a primitive action.

Primitive actions fall into three classes:

- external actions that affect the world;
- mental actions that affect the agent's beliefs;

- control actions that affect the agent's intentions.

External actions are domain specific actions like pickup, putdown etc. in the Blocksworld. The two mental actions that we have are *believing* a proposition, and *disbelieving* a proposition. Our repertoire of control actions includes *sequencing, conditional, iterative, and achieve* actions. A *sequencing* action represents the agent's *intention* to perform its object actions in a given sequence. The *conditional* and *iterative* actions are modeled after Dijkstra's guarded-if and guarded-loop commands respectively [4]. The *achieve* action deduces plans for achieving some proposition and forms the intention of performing one of them. The *conditional* and *iterative* control actions enable the specification of non-linear partial plans. We can also have an explicit representation of non-linear plans using an appropriate control action. We have also designed a control action that can be used for posting constraints on plan variables (as in [27]).

2.3.6 Modeling external effects of actions

As mentioned above, external actions are domain specific actions that affect the outside world. For example, if the agent has an arm and is asked to pick up a block, the arm actually moves to the block, grasps it, and then lifts it up. Depending on the set of interfaces provided to the agent (like an arm, a speech synthesizer, etc.) we need to be able to carry out the action in the external world. This is done by writing Common Lisp functions that access the external interface. For instance, we can model the external effects of the 'say' action by driving a speech synthesizer or by simply printing the message on the screen. The **define-primaction** function enables us to do this. Thus, to model saying something by printing it on the screen, we will have

```
(define-primaction say (n)
  "n is the node representing the act of saying. The node at the
   end of object1 arc is printed. choose.ns and pathfrom are
   SNePS interface functions to access parts of a structured node."
   (format t " ~A " (choose.ns (pathfrom 'object1 n))))
```

Thus when the agent executes the action represented by

```
(build action say object1 FOO)
```

the above code for **say** is executed, resulting in 'FOO' appearing on the screen. How an action gets scheduled to be executed is discussed in the next section. Section 2.6.1 has some more examples of modeling primitive blocksworld actions.

2.4 THE PLANNING PARADIGM

Besides having a current set of beliefs about the world, the system also has beliefs about plans for achieving goals, and about how complex actions can be decomposed into partial plans. The overall architecture of the system is similar to that of the PRS system [9] The acting executive (called an *interpreter* or a *reasoning mechanism* in PRS) manipulates these components. It maintains an acting queue (referred to as a *process stack* in PRS) that contains all the scheduled actions to be performed as a part of some plan and so it represents the system's *intentions*. The system can also form its own intentions in response to changing beliefs. SNIP, the SNePS inference package is used for several tasks: to find plans for complex tasks; as part of the achieve action, to find a plan to achieve some goal; and also as the *truth criterion* (also called the *question answering procedure*, see [5]). Hence, it is used as the *plan decision procedure* in our system. SNIP is implemented on a simulated multi-processing system. In the future, we will be able to do hypothetical reasoning using SNeBR [2] for state-based plan projection.

2.4.1 The acting executive

We want the system to carry out plans, as well as to discuss them, reason about them, and recognize them. Certainly, since the system is currently without eyes, hands, or mobility, its repertoire of primitive actions is small, but, for now, as shown above, we can simulate other actions by appropriate printed messages. SNACTor, the acting system is composed of a queue of acts to be carried out, and an acting executive. The queue of acts represents the system's intentions for carrying out the acts on the queue in that order. Intentions are formed by either an explicit request from a user to do something, or by committing to a plan that needs to be executed to fulfill a complex act or a goal. Explicit requests are made using the **perform** command. For example,

```
(perform (build action say FOO))
```

is an explicit request to the system to say 'FOO'. The act is put on the **act-queue** and the acting executive takes charge. Currently, the acting executive is the following loop:

```
while act-queue is not empty do
      if   the first-act on the act-queue has preconditions
           and they are not currently satisfied
           then insert the achieving of them on the front of the act-queue
      else remove the first-act from the act-queue;
           deduce   effects of first-act,
                    and insert the believing of them on the front
                    of the act-queue;
           if   first-act is primitive
                then perform it
           else deduce plans for carrying out first-act
                (using SNIP and available rules),
                choose one of them,
                and insert it on the front of the act-queue
           end if
      end if
end while
```

Notice that the effects of the act about to be performed are retrieved and scheduled to be believed before the act is actually performed. This guarantees that proper effects of the act are retrieved depending on the context that exists at that time. This flexibility in dynamically determining the effects of acts is what enables us to avoid having multiple operators for the same action.

When preconditions for an act exist and some of them are found not to be true, we schedule the achieving of all of them on the queue. The intention to perform the act is now pushed behind the intention to achieve these preconditions. Once all the preconditions are achieved, and we are ready to perform the act, they are checked again (just in case achieving some precondition renders another one false). Later on, we intend to incorporate critics, that will enable detecting of such conflicts, and more sophisticated reasoning about plans.

From the above loop, it can be seen that at this stage of our work, we are assuming that a plan will be found for every complex act, and that every act will be successful. These assumptions will be removed as we proceed. SNACTor can also be made to do classical planning in case it is not able to find a plan to achieve a goal. This is done in the spirit of STRIPS[7] by reasoning about effects of actions. As mentioned above, SNeBR can be used for hypothetical reasoning.

2.5 SYNTAX AND SEMANTICS OF CONTROL ACTIONS

We are now ready to examine the syntax and operational semantics of our current set of control actions.

Syn. 6: *sequence* ::= ACTION: SNSEQUENCE
OBJECT1: *act1*
OBJECT2: *act2*

This means that a *sequence* act is represented by a node with an ACTION arc to the node, SNSEQUENCE, an OBJECT1 arc to an *act* node, and an OBJECT2 arc to another *act* node.

Sem. 6: *act2* is inserted on the front of the act queue, and then *act1* is inserted in front of it.

For example, a plan to get a block on a support is to pick it up and then put it down on the support. This can be derived using the *plan-goal-proposition* and *snsequence* as

```
(assert forall ($block $support)
        &ant ((build member *block class block)
              (build member *support class object))
        cq (build plan (build action snsequence
                              object1 (build action pickup object1 *block)
                              object2 (build action putdown
                                       object1 *block object2 *support))
                   goal (build rel on arg1 *block arg2 *support)))
```

Since either or both of *act1* and *act2* can themselves be *snsequence* acts, we have a general structure for plans of sequential actions.

Syn. 7: *conditional* ::= ACTION: SNIF
OBJECT1: {CONDITION: *propositioni*
THEN: *acti*}

This means that a *conditional* act is represented by a node with an ACTION arc to the node, SNIF, and OBJECT1 arcs to an arbitrary number of nodes, each with a CONDITION arc to a *proposition* node and a THEN arc to an *act* node.

Sem. 7: If no *proposition* is true, does nothing. Otherwise, arbitrarily chooses one *acti* whose corresponding *propositioni* is true, and puts it on the front of the act queue. (Based on Dijkstra's guarded if [4].)

For example, an act of saying 'HELLO' contingent upon having permission can be expressed as

```
(build action snif
       object1 (build condition (build have permission)
                       then (build action say object1 HELLO)))
```

Syn. 8: *iteration* ::= ACTION: SNITERATE
OBJECT1: {CONDITION: *propositioni*
THEN: *acti*}

Sem. 8: If no *proposition* is true, does nothing. Otherwise, arbitrarily chooses one *acti* whose corresponding *propositioni* is true, and puts on the front of the act queue a sequence whose OBJECT1 is *acti* and whose OBJECT2 is the *iteration* node itself. (Based on Dijkstra's guarded loop [4].)

For example, the act of repeatedly saying 'HELLO' contingent upon having 'hello-permission' and saying 'THERE' contingent upon having 'there-permission' can be expressed as

```
(build action sniterate
       object1 ((build condition (build have hello-permission)
                       then (build action snsequence
                         object1 (build action say object1 HELLO)
                         object2 (build action forget
                                object1 (build have hello-permission))))
                (build condition (build have there-permission)
                       then (build action snsequence
                         object1 (build action say object1 THERE)
                         object2 (build action forget
                                object1 (build have there-permission))))))
```

Syn. 9: *achieve* ::= ACTION: ACHIEVE
OBJECT1: *proposition*

Sem. 9: If *proposition* is true, does nothing. Otherwise, deduces plans for achieving *proposition*, chooses one of them, and puts it on the front of the act queue.

For example, in order to achieve a state in which *BLOCKA is clear* we'll have the act

```
(perform (build action achieve
                object1 (build property clear object BLOCKA)))
```

Thus, we can write plans for achieving goals as well as plans for decomposing a complex act. The domain normally determines the kinds of plans required (*i.e.*, goal-based or act-decomposition based or both). However, as we will see, in the case of the blocksworld, and possibly in other domains, it may become hard to distinguish between something that characterizes a state and something that expresses an act. For example, "Clear BLOCKA" could be interpreted as a command to perform the act of clearing BLOCKA or a goal to

achieve a state in which BLOCKA is clear. We are still exploring this issue. In any case, if required, we can model and use both interpretations.

Other control acts may be defined in the future, in particular a parameterized act that uses a sensory act to identify some object, and then performs some action on the identified object.

2.6 USING PLANS

In this section we will demonstrate how to model actions and do planning in the domain of a blocksworld. We will show two different models—the first model (based on [15]) has four primitive actions (*pickup, putdown, stack,* and *unstack*); the second one has only two (*pickup,* and *putdown*). We will demonstrate the first model through the natural language interface. The second model will be explained in terms of SNePSUL commands so as to give the reader an idea of how to directly use the planning facilities that we have implemented.

2.6.1 The natural language front-end to a blocksworld

The natural language understanding component is implemented in a Generalized ATN grammar and is used for analyzing sentences and for generating English responses. SNACTor begins with an empty knowledge-base. In the role of informant, we interact with SNACTor using English sentences about the domain, instructing SNACTor about the various actions that it can do, and how to solve problems in that domain. The input sentences are analyzed using a domain-specific grammar, the results of which are new beliefs in the knowledge-base. A natural language generation grammar takes the new beliefs and expresses them back in English to show SNACTor's understanding to the informant. Requests to do some action are sent to an acting executive that may then generate and execute a plan to fulfill the request. The informant may also ask questions about plans and the way the system would solve various problems.

As mentioned above, our model of the blocksworld consists of some rectangular blocks on a table. The blocks are identified by names (like blocka, blockb, *etc.*). Beliefs about the current state of the blocksworld are held using the following predicates:

ontable(x) x is on the table.

clear(x) x is clear (*i.e.,* there is nothing on top of it).

on(x,y) x is on top of y.

These are represented in SNePS using the *property-object* proposition case-frames and the *rel-arg1-arg2* proposition case frames.

Syn. 10: *property-object-prop ::=* PROPERTY: *property*
OBJECT: *object*

Sem. 10: *property-object-prop* is a proposition node representing the proposition that *object* has property *property.*

This can be used to represent the **ontable** and **clear** predicates. For example, to assert that BLOCKA is clear and is on the table, we have

```
(assert property clear object BLOCKA)
(assert property ontable object BLOCKA)
```

Syn. 11: *rel-arg1-arg2-prop* ::= REL: *relation*
ARG1: *object1*
ARG2: *object2*

Sem. 11: *rel-arg1-arg2-prop* is a proposition node representing that *relation* holds between *arg1* and *arg2*.

This can be used to represent the on predicate. For example, to assert that BLOCKC is on top of BLOCKA, we have

```
(assert rel on arg1 BLOCKC arg2 BLOCKA)
```

Thus when we tell the system in English

```
Blockc is clear. Blockc is on the table. Blocka is clear.
Blocka is on blockc.
```

the sentences are analyzed by the natural language understanding system and the following SNePSUL commands are executed to assert the beliefs expressed by the sentences

```
(ASSERT PROPERTY (BUILD LEX clear)
        OBJECT (BUILD LEX blockc))
(ASSERT PROPERTY (BUILD LEX ontable)
        OBJECT (BUILD LEX blockc))
(ASSERT PROPERTY (BUILD LEX clear)
        OBJECT (BUILD LEX blocka))
(ASSERT REL (BUILD LEX on)
        ARG1 (BUILD LEX blocka)
        ARG2 (BUILD LEX blockc))
```

The LEX arcs are used by the system to represent that the name of the property (or rel) is expressed in English (*i.e.*, lexically) as "clear" or "on" etc. Thus appropriate morphological analyses and syntheses can be applied to them during understanding and generation. This is discussed in more detail in [22].

The generation component then takes over and expresses the resulting propositions in English, which forms the system's response to demonstrate its understanding of what was said

```
I understand that blockc is clear.
I understand that blockc is ontable.
I understand that blocka is clear.
I understand that blocka is on blockc.
```

Modeling primitive actions

We can ask the agent to perform the following primitive actions

pickup(x) This is an action specifying the agent to pick up **x** where **x** is some block.

putdown(x) This action can be executed when the agent is already holding **x** where **x** is some block. This is a request to put the block on the table.

stack(x,y) This action is a request to put **x** on **y**. **x** and **y** are some specified blocks and **x** is being held.

unstack(x,y) This is a request to pick up **x** from the top of **y**. **x** and **y** are some specified blocks and **x** is on **y**.

The external interface to the blocksworld is simulated in the form of an appropriate message. The system also creates a graphics window that graphically shows the state of the blocksworld at any given instant. The primitive actions are appropriately interfaced so as to simulate their external effects in the graphics window. The following Common Lisp code specifies external effects of these actions:

```
(define-primaction pickup (n)
  "Prints a message to indicate the execution of the pickup
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions. bw-pickup is the graphics interface
   to the blocksworld window. It graphically shows the picking up
   operation in the window."
  (format t "~&Pickup ~A from table.~%"
   (bw-pickup
    (eval (node-na (choose.ns (pathfrom '(object1 lex) (node n))))))))

(define-primaction putdown (n)
  "Prints a message to indicate the execution of the putdown
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions. bw-putdown is the graphics interface
   to the blocksworld window. It graphically shows the putdown
   operation in the window."
  (format t "~&Putdown ~A on table.~%"
   (bw-putdown
    (eval (node-na (choose.ns (pathfrom '(object1 lex) n)))))))

(define-primaction stack (n)
  "Prints a message to indicate the execution of the stack
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions. bw-stack is the graphics interface
```

```
   to the blocksworld window. It graphically shows the stacking
   operation in the window."
  (format t "~&Stack ~A on ~A.~%"
   (bw-stack (eval node-na (choose.ns (pathfrom '(object1 lex) n)))
             (eval node-na (choose.ns (pathfrom '(object2 lex) n))))))

(define-primaction unstack (n)
  "Prints a message to indicate the execution of the unstack
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions. bw-unstack is the graphics interface
   to the blocksworld window. It graphically shows the unstacking
   operation in the window."
  (format t "~&Unstack ~A from ~A.~%"
   (bw-unstack (eval node-na (choose.ns (pathfrom '(object1 lex) n)))
               (eval node-na (choose.ns (pathfrom '(object2 lex) n))))))
```

2.6.2 An annotated example

We will now show how the system is instructed about a blocksworld. We begin by getting into SNePS

```
> (sneps)

   Welcome to SNePS-2.0
   1/10/1989 20:36:47
*
```

At this point we will load SNACTor, the parser, the ATN grammar, the lexicon, and define the required arcs. Now, we are ready to get into the parser and start telling the system about our blocksworld. We begin by informing it about the primitive actions.[1]

```
*(^ (parse -1))

 ATN parser initialization...
 Input sentences in normal English orthographic convention.
May go beyond a line by having a space followed by a <CR>
To exit parser, write ^end.

 : Picking up is a prim         itive action.
 I understand that pickup is a primitive action.
 Time (sec.): 2.15
```

[1]Sentences preceding with a colon (:) are inputs by the user. The rest are all system responses

```
: After picking up a block
        the block is not clear and
        the block is not ontable and
        the block is held.

I understand that after performing pickup on a block, the block
is not clear.

I understand that after performing pickup on a block, the block
is not ontable.

I understand that after performing pickup on a block, the block
is held.
Time (sec.): 5.883333

: Putting down is a primitive action.
I understand that putdown is a primitive action.
Time (sec.): 0.71666664

: After putting down a block
        the block is not held and
        the block is clear and
        the block is ontable.

I understand that after performing putdown on a block, the block
is not held.

I understand that after performing putdown on a block, the block
is clear.

I understand that after performing putdown on a block, the block
is ontable.
Time (sec.): 5.0

: Stacking is a primitive action.
I understand that stack is a primitive action.
Time (sec.): 0.5833333

: After stacking a block on another block
        the latter block is not clear and
        the former block is not held and
        the former block is on the latter block and
```

```
       the former block is clear.

I understand that after performing stack on a block and another
block, the latter block is not clear.

I understand that after performing stack on a block and another
block, the former block is not held.

I understand that after performing stack on a block and another
block, the former block is on the latter block.

I understand that after performing stack on a block and another
block, the former block is clear.
Time (sec.): 7.1833334

: Unstacking is a primitive action.
I understand that unstack is a primitive action.
Time (sec.): 1.8166667

: After unstacking a block from another block
       the former block is not clear and
       the former block is not on the latter block and
       the latter block is clear and
       the former block is held.

I understand that after performing unstack on a block and another
block, the former block is not clear.

I understand that after performing unstack on a block and another
block, the former block is not on the latter block.

I understand that after performing unstack on a block and another
block, the latter block is clear.

I understand that after performing unstack on a block and another
block, the former block is held.
Time (sec.): 6.883333
```

At this point we have successfully told the system about the four primitive actions and their effects. We now tell it about some plans.

```
: If a block is on another block then
       a plan to achieve that the former block is held
       is to achieve that the former block is clear and then
```

```
        unstack the former block from the latter block.

I understand that if a block is on another block then a plan
to achieve that the former block is held is by achieving that
the former block is clear and then performing unstack on the
former block and the latter block.
Time (sec.): 3.9

: If a block is ontable and the block is clear then
        a plan to achieve that the block is held
        is to pick up the block.

I understand that if a block is clear and the block is ontable
then a plan to achieve that the block is held is by performing
pickup on the block.
Time (sec.): 2.9833333

: A plan to achieve that a block is ontable
        is to achieve that the block is held and then
        put down the block.

I understand that a plan to achieve that a block is ontable is
by achieving that the block is held and then performing putdown
on the block.
Time (sec.): 3.4333334

: A plan to achieve that a block is on another block
        is to achieve that the latter block is clear and then
        achieve that the former block is held and then
        stack the former block on the latter block.

I understand that a plan to achieve that a block is on another
block is by achieving that the latter block is clear and then
achieving that the former block is held and then performing stack
on the former block and the latter block.
Time (sec.): 4.133333

: If a block is on another block then
        a plan to achieve that the latter block is clear
        is to achieve that the former block is clear and then
        achieve that the former block is ontable.
```

```
I understand that if a block is on another block then a plan
to achieve that the latter block is clear is by achieving that
the former block is clear and then achieving that the former
block is ontable.
Time (sec.): 3.7

: A plan to pile a block on another block on a third block
       is to achieve that the third block is ontable and then
       achieve that the second block is on the third block and then
       achieve that the first block is on the second block.

I understand that a plan for performing pile on a block and another
block and another block is by achieving that the third block
is ontable and then achieving that the second block is on the
third block and then achieving that the first block is on the
second block.
Time (sec.): 5.383333
```

Now the system knows plans for decomposing some complex acts (like piling) as well as plans for achieving some goals. We are now ready to describe a state of blocks in the blocks world to play with.

```
: blockc is clear. blockc is ontable.
I understand that blockc is clear.
I understand that blockc is ontable.
Time (sec.): 0.65

: blockb is clear. blockb is ontable.
I understand that blockb is clear.
I understand that blockb is ontable.
Time (sec.): 1.68333334

: blocka is clear. blocka is ontable.
I understand that blocka is clear.
I understand that blocka is ontable.
Time (sec.): 0.6666667
```

We now ask it to do some simple things.

```
: pick up blockc.
I understand that you want me to perform the action of pickup
on blockc.

Pickup blockc from table.
```

```
Disbelieve blockc is clear. Disbelieve blockc is ontable.
Believe blockc is held.
 Time (sec.): 4.0666666

 : stack blockc on blocka.
 I understand that you want me to perform the action of stack
 on blockc and blocka.

Stack blockc on blocka.
Believe blockc is clear. Disbelieve blocka is clear.
Disbelieve blockc is held. Believe blockc is on blocka.
 Time (sec.): 5.45

 : pick up blockb
 I understand that you want me to perform the action of pickup
 on blockb.

Pickup blockb from table.
Disbelieve blockb is clear. Disbelieve blockb is ontable.
Believe blockb is held.
 Time (sec.): 5.2833333

 : put down blockb.
 I understand that you want me to perform the action of putdown
 on blockb.

Putdown blockb on table. Disbelieve blockb is held.
Believe blockb is clear. Believe blockb is ontable.
 Time (sec.): 5.5833335
```

blockc is on blocka and blockb is sitting on the table. We now ask it to pile blocka on blockb on blockc.

```
 : pile blocka on blockb on blockc.
 I understand that you want me to perform the action of pile on
 blocka and blockb and blockc.
```

SNACTor now goes into its acting executive which realizes that piling is a complex act and so needs to be decomposed. This is where it uses beliefs acquired from our earlier dialog and finds decompositions:

```
A plan to pile blocka on blockb on blockc is to achieve that blockc
is on the table and then achieve that blockb is on blockc and then
achieve that blocka is on blockb.
```

And this decomposition continues depth-first until it finds an appropriate action to execute.

```
Want to achieve blockc is ontable.
Want to achieve blockc is held.
Want to achieve blockc is clear. Already Achieved.

Unstack blockc from blocka. Disbelieve blockc is on blocka.
Believe blocka is clear. Disbelieve blockc is clear.
Believe blockc is held.

Putdown blockc on table. Believe blockc is clear.
Believe blockc is ontable. Disbelieve blockc is held.

Want to achieve blockb is on blockc.
Want to achieve blockc is clear. Already Achieved.
Want to achieve blockb is held.

Pickup blockb from table. Disbelieve blockb is clear.
Disbelieve blockb is ontable. Believe blockb is held.

Stack blockb on blockc. Disbelieve blockb is held.
Believe blockb is on blockc. Believe blockb is clear.
Disbelieve blockc is clear.

Want to achieve blocka is on blockb.
Want to achieve blockb is clear. Already Achieved.
Want to achieve blocka is held.

Pickup blocka from table. Believe blocka is held.
Disbelieve blocka is ontable. Disbelieve blocka is clear.

Stack blocka on blockb. Believe blocka is on blockb.
Disbelieve blocka is held. Believe blocka is clear.
Disbelieve blockb is clear.
 Time (sec.): 311.25
```

2.6.3 Discussing plans

We have seen how we can instruct SNACTor about planning in a domain and how we can describe situations to it and subsequently ask it to do things by using the plans it derives. We can also ask questions about plans and the various beliefs that it holds. For example,

```
 : Is blocka on blockb?
```

```
Yes, blocka is on blockb.
Time (sec.): 1.8

: Is blocka ontable?
No, blocka is not ontable.
Time (sec.): 0.8333333

: Is blocka on blockc?
I really don't know if blocka is on blockc.
Time (sec.): 0.96666664

: How would you pile a block on another block on another block?
I understand that a plan for performing pile on a block and another
block and another block is by achieving that the third block
is ontable and then achieving that the second block is on the
third block and then achieving that the first block is on the
second block.
Time (sec.): 14.05
```

SNIP, the plan decision procedure, is used to derive an appropriate plan and respond to the query.

2.6.4 Another Example

In this example we will present yet another model of the blocksworld. This time we will do away with *stack* and *unstack* primitive actions. We will utilize context-dependent effects as well as preconditions in modeling these actions. This time we will give SNePSUL code instead of natural language dialog. This will also give the reader a better understanding of how to use our representations.

The set of beliefs about the state of the blocksworld will be represented using clear and on predicates. They will be used as defined above. We will treat table as just another object. We will only have pickup and putdown primitive actions.

Modeling pickup

The external effects of pickup are modeled by the function

```
(define-primaction pickup (n)
  "Prints a message to indicate the execution of the pickup
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions."
  (format t "~&Pickup ~A.~%"
          (choose.ns (pathfrom '(object1 lex) (node n)))))
```

Since Pickup is a primitive action, we can assert

```
(assert member (build lex "pickup") class "primitive")
```

and now we go on to describe the preconditions of pickup. The only precondition for picking up a block is to make sure that the block is clear. Thus

```
(assert forall $block
        ant (build member *block class (buile lex "block"))
        cq (build act (build action (build lex "pickup") object1 *block)
                  precondition (build property (build lex "clear")
                                       object *block)))
```

From now on, everytime a pickup is to be performed, the system will check if the block to be picked up is clear, if not, the acting executive will schedule the act of achieving that it is clear before it is picked up. Effects of picking up a block are:

```
;#1: it is no longer clear
(assert forall *block
        ant (build member *block class (build lex "block"))
        cq (build act (build action (build lex "pickup") object1 *block)
                  effect (build min 0 max 0
                                arg (build property (build lex "clear")
                                           object *block))))

;#2: it is no longer on whatever was supporting it
(assert forall (*block $support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object"))
              (build rel (build lex "on") arg1 *block arg2 *support))
        cq (build act (build action (build lex "pickup") object1 *block)
                   effect (build min 0 max 0
                                 arg (build rel (build lex "on")
                                            arg1 *block
                                            arg2 *support))))

;#3: and its support is now clear.
(assert forall (*block $support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (buile lex "object"))
              (build rel (build lex "on") arg1 *block arg2 *support))
        cq (build act (build action (build lex "pickup") object1 *block)
                   effect (build property (build lex "clear")
                                 object *support)))
```

```
;#4: the block is being held
(assert forall *block
        ant (build member *block class (build lex "block"))
        cq (build act (build action (build lex "pickup") object1 *block)
                  effect (build property (build lex "holding")
                                object *block)))
```

Notice that rules 2 and 3 are context-dependent effects.

Modeling putdown

The external effects of putdown are:

```
(define-primaction putdown (n)
  "Prints a message to indicate the execution of the putdown
   primitive action. choose.ns, pathfrom, node, and node-na are
   SNePS node-access functions."
  (format t "~&Putdown ~A on ~A.~%"
          (choose.ns (pathfrom '(object1 lex) n))
          (choose.ns (pathfrom '(object2 lex) n))))
```

Before putting down a block on a support we have to make sure that we are holding the block and that the support is clear. Hence the preconditions for putdown are:

```
;#1: Make sure the block is being held
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                object1 *block object2 *support)
                  precondition (build property (build lex "holding")
                                       object *block)))

;#2: Make sure the support is clear
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                object1 *block object2 *support)
                  precondition (build property (build lex "clear")
                                       object *support)))
```

And finally, after putting down a block on some support, we have the following effects:

```
;#1: The block is no longer being held
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                    object1 *block object2 *support)
                      effect (build min 0 max 0
                                    arg (build property (build lex "holding")
                                                        object *block))))

;#2: The block is clear
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                    object1 *block object2 *support)
                      effect (build property (build lex "clear")
                                    object *block)))

;#3: The block is on the support
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                    object1 *block object2 *support)
                      effect (build rel (build lex "on")
                                    arg1 *block arg2 *support)))

;#4: The support is not clear
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build act (build action (build lex "putdown")
                                    object1 *block object2 *support)
                      effect (build min 0 max 0
                                    arg (build property (build lex "clear")
                                                        object *support))))
```

Plans

We will write rules for plans for doing the same kind of things that we did in the earlier example. However, here we will notice that the plans are much simpler and more general. For example, to achieve the goal of holding a block, we simply ask it to pick it up.

```
;#1: Plan rule for holding a block
(assert forall *block
        ant (build member *block class (build lex "block"))
        cq (build plan (build action (build lex "pickup")
                                     object1 *block)
                       goal (build property (build lex "holding")
                                     object *block)))
```

To get a block on a supporting object a general plan is to pick it up and then put it down on the object.

```
;#2: Plan rule for getting a block on a support
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object")))
        cq (build plan (build action (build lex "snsequence")
                                     object1 (build action (build lex "pickup")
                                                           object1 *block)
                                     object2 (build action (build lex "putdown")
                                                           object1 *block
                                                           object2 *support))
                       goal (build rel (build lex "on")
                                     arg1 *block arg2 *support)))
```

If we need to clear a block that is a support for another block then a plan to do it is to first pick up the top block and then put it down on the table.

```
;#3: Plan rule for clearing a support
(assert forall (*block *support)
        &ant ((build member *block class (build lex "block"))
              (build member *support class (build lex "object"))
              (build rel (build lex "on") arg1 *block arg2 *support))
          cq (build plan (build action (build lex "snsequence")
                                       object1 (build action (build lex "pickup")
                                                             object1 *block)
                                       object2 (build action (build lex "putdown")
                                                             object1 *block
                                                             object2 (build lex "the-table")))
                         goal (build property (build lex "clear")
                                       object *support)))
```

Finally, to build a pile of three blocks, we first put the third one on the table, then the second one on the third one, and then the first one on the second one.

```
;#4: Plan rule for piling three blocks
```

```
(describe
 (assert forall (*block *support $third-block)
         &ant ((build member *block class (build lex "block"))
               (build member *support class (build lex "block"))
               (build member *third-block class (build lex "block")))
         cq (build act (build action pile object1 *third-block
                                          object2 *block
                                          object3 *support)
                       plan (build action (build lex "snsequence")
                                  object1 (build action (build lex "achieve")
                                                 object1 (build rel (build lex "on")
                                                                    arg1 *third-block
                                                                    arg2 "the-table"))
                                  object2 (build action (build lex "snsequence")
                                  object1 (build action (build lex "achieve")
                                                        object1 (build rel (build lex "on")
                                                                           arg1 *support
                                                                           arg2 *third-block))
                                  object2 (build action (build lex "achieve")
                                                        object1 (build rel (build lex "on")
                                                                           arg1 *block
                                                                           arg2 *support))))))
```

Annotated example

We will describe a state of the blocksworld, we begin by getting into SNePS.

```
> (sneps)

   Welcome to SNePS-2.0
   1/11/1989 13:46:26
*
```

At this point we load the Lisp definitions of the primitive actions and the rules described above and assertions about which actions are primitive. In what follows, we describe a state in which the blocks blocka, blockb, and blockc are clear and on the table.

```
(describe (assert property (build lex "clear")
                  object (build lex "blocka")))
(M29! (OBJECT (M28 (LEX blocka))) (PROPERTY (M10 (LEX clear))))
 CPU time : 0.20     GC time : 0.00

(describe (assert rel (build lex "on")
                  arg1 (build lex "blocka")
```

```
                arg2 (build lex "the-table")))
(M30! (ARG1 (M28 (LEX blocka))) (ARG2 (M26 (LEX the-table)))
      (REL (M11 (LEX on))))
 CPU time : 0.15     GC time : 0.00

(describe (assert property (build lex "clear")
                object (build lex "blockb")))
(M32! (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
 CPU time : 0.17     GC time : 0.00

(describe (assert rel (build lex "on")
                arg1 (build lex "blockb")
                arg2 (build lex "the-table")))
(M33! (ARG1 (M31 (LEX blockb))) (ARG2 (M26 (LEX the-table)))
      (REL (M11 (LEX on))))
 CPU time : 0.13     GC time : 0.00

(describe (assert property (build lex "clear")
                object (build lex "blockc")))
(M35! (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
 CPU time : 0.17     GC time : 0.00

(describe (assert rel (build lex "on")
                arg1 (build lex "blockc")
                arg2 (build lex "the-table")))
(M36! (ARG1 (M34 (LEX blockc))) (ARG2 (M26 (LEX the-table)))
      (REL (M11 (LEX on))))
 CPU time : 0.22     GC time : 0.00
```

Now, we ask SNACTor to achieve a state in which blockb is on blockc.

```
(perform (build action (build lex "achieve")
               object1 (build rel (build lex "on")
                                  arg1 (build lex "blockb")
                                  arg2 (build lex "blockc"))))
```

The acting executive acknowledges this by printing:

```
Want to achieve:
(M37 (ARG1 (M31 (LEX blockb))) (ARG2 (M34 (LEX blockc)))
     (REL (M11 (LEX on))))
```

At this point it has found a plan using the plan rule 2 described above. Thus it is going to try and pick up blockb and put it down on blockc. Before performing pickup on blockb, it

has to find if there are any preconditions and whether they are satisfied:[2]

```
Preconditions...
(M32! (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Are satisfied.

Pickup blockb.
Disbelieve:
(M32 (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Believe:
(M46! (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
Disbelieve:
(M33 (ARG1 (M31 (LEX blockb))) (ARG2 (M26 (LEX the-table)))
     (REL (M11 (LEX on))))
Believe:
(M50! (OBJECT (M26 (LEX the-table))) (PROPERTY (M10 (LEX clear))))
```

Since the precondition for pick up, *i.e.*, blockb is clear was satisfied, it picked up blockb and asserted the effects of doing that. Now it is ready to execute the second part of the plan, *i.e.* put down blockb on blockc. Once again, it will derive and test the predconditions for putdown. Since they are satisfied, it will go ahead and complete the plan.

```
Preconditions...
(M35! (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
(M46! (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
Are satisfied.

Putdown blockb on blockc.
Believe:
(M32! (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Believe:
(M37! (ARG1 (M31 (LEX blockb))) (ARG2 (M34 (LEX blockc)))
      (REL (M11 (LEX on))))
Disbelieve:
(M46 (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
Disbelieve:
(M35 (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
 CPU time : 50.30     GC time : 0.00
```

Let us now ask it to put blocka on blockc. Since blockb is sitting on top of blockc, it will first have to take care of that. Let us give the command

[2]An exclamation (!) in front of the node number indicates that it currently believes the proposition represented by that node.

```
(preform (build action (build lex "achieve")
                object1 (build rel (build lex "on")
                                   arg1 (build lex "blocka")
                                   arg2 (build lex "blockc"))))

Want to achieve:
(M68 (ARG1 (M28 (LEX blocka))) (ARG2 (M34 (LEX blockc)))
     (REL (M11 (LEX on))))
```

Once again it has found the same plan as before; *i.e.*, first it should pick up blocka and then put it down on blockc. Since blocka is clear it will pick it up and assert its effects:

```
Preconditions...
(M29! (OBJECT (M28 (LEX blocka))) (PROPERTY (M10 (LEX clear))))
Are satisfied.

Pickup blocka.
I already believe:
(M50! (OBJECT (M26 (LEX the-table))) (PROPERTY (M10 (LEX clear))))
Disbelieve:
(M29 (OBJECT (M28 (LEX blocka))) (PROPERTY (M10 (LEX clear))))
Believe:
(M77! (OBJECT (M28 (LEX blocka))) (PROPERTY (M12 (LEX holding))))
Disbelieve:
(M30 (ARG1 (M28 (LEX blocka))) (ARG2 (M26 (LEX the-table)))
     (REL (M11 (LEX on))))
```

It has completed the first step of the plan and is now ready to put down blocka on blockc. However, blockc is not clear since it has blockb sitting on it. Thus it has to take care of that situation:

```
Preconditions...
(M35 (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
(M77! (OBJECT (M28 (LEX blocka))) (PROPERTY (M12 (LEX holding))))
Are not satisfied.
```

It notices that blockc is not clear, so before it puts blocka down on blockc it has to remove blockb from the top of blockc. It creates a new intention to clear blockc.

```
Want to achieve:
(M35 (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
```

It uses plan rule 3 described above. Thus now it has to first pick up blockb, put it down on the table, and then put down blocka on blockc.

```
Preconditions...
(M32! (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Are satisfied.

Pickup blockb.
Believe:
(M35! (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
Disbelieve:
(M32 (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Believe:
(M46! (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
Disbelieve:
(M37 (ARG1 (M31 (LEX blockb))) (ARG2 (M34 (LEX blockc)))
     (REL (M11 (LEX on))))

Preconditions...
(M46! (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
(M50! (OBJECT (M26 (LEX the-table))) (PROPERTY (M10 (LEX clear))))
Are satisfied.

Putdown blockb on the-table.
Believe:
(M32! (OBJECT (M31 (LEX blockb))) (PROPERTY (M10 (LEX clear))))
Believe:
(M33! (ARG1 (M31 (LEX blockb))) (ARG2 (M26 (LEX the-table)))
      (REL (M11 (LEX on))))
Disbelieve:
(M46 (OBJECT (M31 (LEX blockb))) (PROPERTY (M12 (LEX holding))))
```

Now that it has gotten rid of blockb, it is now ready to put down blocka on blockc. This time all the preconditions will be found to be satisfied and the plan will conclude successfully. Since we haven't put any restrictions on how many blocks it can hold (this does make things simpler) it did not put blocka down before picking up blockb.

```
Want to achieve:
(M77! (OBJECT (M28 (LEX blocka))) (PROPERTY (M12 (LEX holding))))
Already Achieved.

Preconditions...
(M35! (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
(M77! (OBJECT (M28 (LEX blocka))) (PROPERTY (M12 (LEX holding))))
Are satisfied.

Putdown blocka on blockc.
```

```
Disbelieve:
(M77 (OBJECT (M28 (LEX blocka))) (PROPERTY (M12 (LEX holding))))
Believe:
(M29! (OBJECT (M28 (LEX blocka))) (PROPERTY (M10 (LEX clear))))
Disbelieve:
(M35 (OBJECT (M34 (LEX blockc))) (PROPERTY (M10 (LEX clear))))
Believe:
(M68! (ARG1 (M28 (LEX blocka))) (ARG2 (M34 (LEX blockc)))
      (REL (M11 (LEX on))))
 CPU time : 394.40     GC time : 0.00
```

2.7 STRATEGIC PLANNING FRAMEWORK

The current research of the U. Mass. group concerns the comprehension of the following paragraph:

> Nancy asked Tom if an inanimate object, such as a table, can exert a force. Tom said he didn't think so. Nancy pointed to a pile of books on the table and asked if the table exerts an upward force on the books. Tom said no. Nancy placed the books on Tom's hand. Tom had to exert a great force to keep the books steady. Nancy asked Tom to compare the two situations. Tom said the table must also have exerted a force on the books.

The goal is to comprehend the paragraph, answer questions about it and recognize its underlying plans. The system should answer questions such as:

> **Q:** Why is Nancy asking Tom about books on the table?
> **A:** Nancy created a specific example of the topic in order to find out if Tom understood it.
>
> **Q:** Why did Nancy place the books on Tom's hand?
> **A:** Nancy used Tom's hand as an anchor example. After she asked about an abstract concept, she provided an instantiation of the concept that gave Tom direct experience of the topic.

Working with SUNY/Buffalo, the U. Mass group has sucessfully ported the SNePS-2 knowledge representation, inference, and parsing systems to UMASS. We have developed domain knowledge and inference rules in SNePS-2 to support comprehension of the above paragraph and have represented discourse and tutoring knowledge in terms of acts, preconditions, goals, primitive actions, and complex actions.

For example, we have represented typical acts such as *Ask-Question* and *Answer-Question* and typical plans such as *Teach-by-Leading-Question* and *Teach-by-Analogy.* Plans are used to represent overarching tutoring strategies, such as *Easier material will be presented before harder material* and *Examples are typically useful* These generalized plans are then used by the system to make top-down expectations about subsequent actions, to infer the occurrence of unstated actions, and to explain actions which serve to satisfy goals.

Discourse and tutoring plans that guide human speakers are generally known to both discussants. For example, a student can infer that a question is being asked as part of the teacher's effort to teach a new or related concept. Plan recognition enables the discussants to understand the underlying intent and nature of the dialogue.

Reactive Planning Framework However, strategic planning is computationally intractable, often producing a exponential search spaces. It places significant constraints on the order of actions within a plan and makes clear predictions about later activities. For example, plan recognition can succeed only if actions are clearly described in terms of how they accomplish goals and subgoals. In fact, dialogue, and especially tutoring dialogues, are frequently characterized by actions that do not satisfy goals. For example, an action might be performed to achieve a goal, and that action might be thwarted causing the goal to be abandoned. 'Real-life' dialogues are complex, uncertain, and immediate. They manifest a kind of unpredictability that requires reactive planning.

In reactive planning domain knowledge is used to react to circumstances, rather than using prestored plan to respond to events. Reactive planning is used in situations in which the world does unexpected things, such as in a dialogue between two agents or when the system can not anticipate a user's action in response to its own actions. It is also used when exceptions exist causing plans to be rewritten, regenerated, or discarded. For example, dialogue must allow for mixed-initiative and thus for interruptions and digression on the part of the user.

The reactive planner employs deep knowledge of the domain, discourse strategies, and tutoring actions to enhance its reasoning ability. Consider, for example, the lexical and syntactic ambiguity in following sentence:

> The spring is supported by the weight.

Here domain and situation knowledge is needed to comprehend the context. For example, if the speaker was previously involved in a tutoring lesson about physics, the system would not interpret the word 'spring' as it might if the speaker was previously describing a country setting complete with river and springs.

The hybrid system uses strategic plans to provide deep expectation of the actions and goals in the text. It uses reactive planning to help understand subtle points in the text. The two planning frameworks compliment each other, supplementing and checking the expectations provided by the other. In combination they describe how discourse is managed and controlled in tutoring.

Our current research focus is to enhance our representation of domain and tutoring knowledge in such a way that it will will augment the system's reactive planning capability.

Model Anaphora We have focused on a third research issue this year, namely Model Anaphora [23]. Such anaphora are distinguished from demonstrative pronouns, (*e.g.*, him, her, and it) which refer to discourse segments in that they are the determination of the reference of words such as "situation". Model anaphora have much richer semantics than do pronoun anaphora.

The larger Model Anaphora are handled through use of a focus stack and a backward search through a set of candidates describing the state of affairs of objects and people in the narrative.

We can use the tutoring paragraph above to illustrate Model Anaphora. In the paragraph Nancy definitively sets the topic of the paragraph with her first question to Tom. This places

"inanimate objects" and "force" in focus, as well as marking them as belonging to the topic. Because of our domain knowledge of physics, we know that her second question, which is more specific, is subsumed by her first, and does not change the topic. When Nancy refers to "the two situations" without supplying a situation index, either "inanimate objects" or "force" is probably the index she has in mind.

Supposing we take "force" as the situation index, we search in the Model for Model Objects that match it. We find the first situation because "push" is a kind of force and Nancy has asked Tom whether the books push up on the table. Hence, *the books on the table* is a situation. The second situation is found because force is explicitly mentioned, hence *the books on Tom's hand* is also a situation. Although force is mentioned in the first sentence, "whether an inanimate object, such as a table, can exert a force," this is discarded as a situation because it lacks concreteness.

We are presently implementing the two phases of processing Model Anaphora described above. Specifically, we first find a situation index, defined as the key that picks the situation out of the Model. Then, we search the focus stack for occurrences of the situation index and determine whether any of these is part of a situation.

Situations are commonly referenced with phrases such as "the situation with X" or "the X situation." These modifying phrases include the *Situation Index* for each of these Situations. A *Situation Index* is a key that picks the situation out of the Model, distinguishing foreground from background, so to speak.

The situation index is usually some common or unifying aspect of the situation. Indeed, the situation index need not be explicitly mentioned in the text. We conclude that elements in focus, regardless of explicit mention in the text, are what are available as situation indexes.

The architecture takes references that make explicit mention of the situation index to be the canonical case; it processes other references by first determining the situation index from the context, thereby reducing them to the canonical case. Thus, Model Anaphora is a two-step process:

1. **Determine the situation index.** Where this is explicitly mentioned, the determination is trivial. Otherwise, the situation index must be computed from the context.

2. **Using the situation index, search the stack of focus spaces and look for matches.** The matching Model Objects will yield a set of candidates, some of which will be discarded because they are not states of affairs or because they lack concreteness or tension. If this does not reduce the set of candidates to a single situation, we choose the most recent candidate.

We claim that situations are states of affairs that exhibit concreteness and tension, and that the process of referring to them involves a situation index. We believe that Model Anaphora is a phenomenon common to a large class of nouns, including "case," "disaster," and "example." These nouns differ somewhat in their semantics, but they all make individuation references to sets of Model Objects.

2.8 PLAN RECOGNITION

In this section we will briefly look at the beginning of a plan recognition component based on our representations. The initial idea that we are exploring is based on the observation that if an actor is performing an act that when we perform it we are are in the process of executing some plan, the actor could possibly be performing the act as a part of a similar plan. Expressed more clearly, we have the rule

if an actor x performs an act $a1$,
  and $a1$ is a PLAN-COMPONENT of a proposition p
  then if $a2$ is the ACT of p
        then x may be engaged in carrying out $a2$
  and if g is a GOAL of a proposition p
        then x may be trying to achieve g.

We can express the above rule using SNePS path-based inference [17, 24] as:

(define-path PLAN-COMPONENT
  (compose PLAN
    (kstar (or (compose (kstar OBJECT2) (or OBJECT1 OBJECT2))
      (compose OBJECT1 THEN)))))

This defines a virtual arc PLAN-COMPONENT to be one that goes from a *plan-act-proposition* or a *plan-goal-proposition* to every act within the plan. For example, the following SNePS node represents that the act of greeting someone (**give-greetings**) can be accomplished by a plan to repeatedly say 'HELLO THERE' thus using the act defined in Section 2.3.5 above.

```
(M36! (ACT GIVE-GREETINGS)
 (PLAN
  (M32 (ACTION (M6 (LEX SNITERATE)))
   (OBJECT1
    (M27 (CONDITION (M21 (LEX PERMISSION)))
     (THEN
      (M26 (ACTION (M1 (LEX SNSEQUENCE)))
           (OBJECT1 (M18 (ACTION (M10 (LEX SAY))) (OBJECT1 HELLO)))
       (OBJECT2 (M24 (ACTION (M16 (LEX FORGET))) (OBJECT1 (M21)))))))
    (M31 (CONDITION (M28 (LEX PERMISSION2)))
     (THEN
      (M30 (ACTION (M1)) (OBJECT1 (M19 (ACTION (M10)) (OBJECT1 THERE)))
       (OBJECT2 (M29 (ACTION (M16)) (OBJECT1 (M28)))))))))))
```

Next we give the plan recognition rule to SNACTor. This rule says that if someone is doing an act which is part of some plan, assume that that person is engaged in the plan. Thus we have

```
(assert forall ($agent $reported-act $planned-act)
        &ant ((build agent *agent act *reported-act)
              (build plan-component *reported-act act *planned-act))
        cq (build agent *agent act *planned-act))
```

Now we tell the system that John performed the act of saying 'HELLO'.

```
(add agent john act (build action say object1 HELLO))
```

Now, let us ask about the act(s) that John performed

```
(describe (deduce agent john act $johns-acts))
(M38! (ACT (M18 (ACTION (M10 (LEX SAY))) (OBJECT1 HELLO))) (AGENT JOHN))
(M52! (ACT GIVE-GREETINGS) (AGENT JOHN))
 CPU time : 6.65     GC time : 0.00
```

As we can see, it comes back with a response saying that John is performing the acts of saying 'HELLO' as well as **give-greetings**. We do not yet have a way of dealing with "may be engaged in" nor with "may be trying to achieve," but this rule indicates our approach to plan recognition within the design of the planning/acting SNePS component described in this report.

2.9 DISCUSSION

Our goal is to model a rational cognitive agent whose behavior is driven by its beliefs, desires, and intentions. We want our agent to do natural language understanding, reason about beliefs, act rationally based on its beliefs, do plan recognition, and plan based text generation. Doing all these tasks in a single coherent framework poses several constraints. We are discovering that SNePS and its underlying theories contribute effectively towards our goal. We have designed and implemented intensional propositional representations for plans. This is a major advancement over operator-based descriptions of plans. Operator-based formulations of actions tend to alienate the discussion of operators themselves. Operators are usually specified in a different language than that used for representing beliefs about states. Moreover, plans (or procedural networks) constructed from these operators can only be accessed by specialized programs (critics, executors) and, like operators, are represented in still another formalism. Our representations for acts, actions, goals, and plans build upon and add to the intensional propositional representations of SNePS. This framework enables us to tackle various tasks mentioned above in a uniform and coherent fashion.

Our current system is being advanced in several directions. In the context of planning, there are issues associated with conjunctive goals[26], non-linear plans [16, 25, 5], and dealing with the effects of actions. As mentioned in [6] explicitly specifying the disbelieving of propositions as a result of performing some action is not natural. We propose to use belief revision (SNeBR) to detect inconsistencies after asserting the effects of an action.

Language used in planning contexts, is slightly more constrained than in arbitrary discourse. Sentences describing plans tend to be declarative, with a syntactically decomposable structure involving goal, effect, and plan definition. Handling reference is simplified by the assumption that common noun phrases correspond to typed variables. Indefinite noun phrases introduce new variables, definite noun phrases refer to previously introduced variables. Natural language generation of plans and rules involves careful selection of relevant attributes of these variables.

Our representation of acts and actions was partially motivated by our goals of doing plan recognition. We are just starting work in this direction[20]. Some preliminary work on plan-based natural language generation is reported in[20].

2.10 CONCLUSION

In this report, we have described the design, and aspects of the implementation, of an intensional representation for plans. These representations have been constrained by issues in cognitive modeling, belief representation, reasoning, and natural language understanding. Plans are structured individual mental concepts, consisting of a structure of acts. Acts are structured individual mental concepts of an action process independent of actor and time. Actions are primitive or complex and fall into three classes—external, mental, and control. The system models intentionality with a queue of acts, and may form new intentions based on its current belief status. Currently, we have an implementation of a Blocksworld involving natural language dialogues about plans, planning and execution of Blocksworld plans, and some examples of plan recognition.

Bibliography

[1] M. J. Almeida. *Reasoning About the Temporal Structure of Narratives.* PhD thesis, Department of Computer Science, SUNY at Buffalo, Buffalo, NY, 1987. Technical Report No. 87-10.

[2] J. ao P. Martins and S. C. Shapiro. Hypothetical reasoning. In *Applications of Artificial Intelligence to Engineering Problems: Proceedings of The 1st International Conference*, pages 1029–1042. Springer-Verlag, Berlin, 1986.

[3] J. ao P. Martins and S. C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35(1):25–79, May 1988.

[4] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] M. Drummond and A. Tate. Ai planning: A tutorial and review. Technical Report AIAI-TR-30, Artificial Intelligence Applications Institute, University of Edinburgh, Edinburgh, November 1987.

[6] M. E. Drummond. A representation of action and belief for automatic planning systems. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 189–212, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[7] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5:189–208, 1971.

[8] M. P. Georgeff. Planning. In *Annual Reviews of Computer Science Volume 2*, pages 359–400. Annual Reviews Inc., Palo Alto, CA, 1987.

[9] M. P. Georgeff. An embedded reasoning and planning system. In J. Webber, J. Tenenberg, and J. Allen, editors, *Advance Proceedings of The Rochester Planning Workshop–From Formal Systems to Practical Systems*, pages 79–101, October 1988.

[10] D. Kumar, S. S. Ali, and S. C. Shapiro. Discussing, Using, and Recognizing Plans in SNePS–Preliminary Report. In P. V. S. Rao and P. Sadanandan, editors, *Modern Trends in Information Technology—Proceedings of the Seventh Biannual Convention of South East Asia Regional Computer Confederation (SEARCC88)*, pages 177–182, New Delhi, India, 1988. Tata McGraw-Hill Publishing Company.

[11] A. L. Lansky. A representation of parallel activity based on events, structure, and causality. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 123–160, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[12] V. Lifschitz. On the semantics of STRIPS. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 1–10, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[13] D. P. McKay and S. C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Los Altos, CA, 1981. Morgan Kaufmann.

[14] E. J. Morgado and S. C. Shapiro. Believing and acting–a study of meta-knowledge and meta-reasoning. In *Proceedings of Encontro Portugues de Inteligencia Artificial (EPIA)*, Oporto, Portugal, September 1985.

[15] N. J. Nilsson. *Principles Of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[16] E. D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North Holland, New York, NY, 1977.

[17] S. C. Shapiro. Path-based and node-based inference in semantic networks. In D. Waltz, editor, *Tinlap–2: Theoretical Issues in Natural Languages Processing*, pages 219–225, New York, 1978. ACM.

[18] S. C. Shapiro. The SNePS semantic network processing system. In N. V. Findler, editor, *Associative Networks: The Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[19] S. C. Shapiro. Generalized augmented transition network grammars for generation from semantic networks. *The American Journal of Computational Linguistics*, 8(1):12–25, 1982.

[20] S. C. Shapiro. Representing plans and acts. In J. W. Esch, editor, *Proceedings of the Third Annual Workshop on Conceptual Graphs*, St. Paul, MN, August 1988. Sponsored by AAAI.

[21] S. C. Shapiro, J. ao P. Martins, and D. P. McKay. Bi-directional inference. In *Proceedings of the Fourth Annual Meeting of the Cognitive Science Society*, pages 90–93, Ann Arbor, MI, 1982.

[22] S. C. Shapiro and W. J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 263–315. Springer–Verlag, New York, 1987.

[23] P. Sibun, S. Anderson, D. Forster, and B. Woolf. Reference to situation. Submitted to the International Joint Conference on Artificial Intelligence, 1989.

[24] R. Srihari. Combining path-based and node-based reasoning in SNePS. Technical Report 183, Department of Computer Science, SUNY at Buffalo, 1981. 52pp.

[25] A. Tate. Generating project networks. In *Proceedings 5th IJCAI*, pages 888–93, 1977.

[26] R. Waldinger. *Achieving Several Goals Simultaneously*, pages 94–136. Ellis Horwood, Chichester, England, 1977.

[27] D. E. Wilkins. *Practical Planning–Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, Palo Alto, CA, 1988.

2.11 TRIPS FUNDED BY RADC

SUNY at Buffalo, Department of CS, February 14–19, 1988, to work with the Natural Language Planning research group: Shapiro.

SUNY at Buffalo, Department of CS, March 10–11, 1988, to work with the Natural Language Planning research group: Shapiro.

First Annual CUNY Conference on Sentence Processing, New York, NY, March 24–27, 1988: Anderson.

SUNY at Buffalo, Department of CS, April 11–15, 1988, to work with the Natural Language Planning research group: Shapiro.

NAIC Meeting, Rockwell International, Syracuse, NY, June 7, 1988: Shapiro.

26th Annual Meeting of the Association for Computational Linguistics, SUNY at Buffalo, and three meetings between the SUNYAB and U.Mass. researchers, June 7–10, 1988: Shapiro, Woolf, Ali, Kumar, Sibun, Forster, Anderson.

RADC Meeting, Minnowbrook, NY, August 9-10, 1989: Shapiro Woolf, Anderson.

NAIC Committee Meeting, Syracuse University, September 27, 1988: Shapiro.

NAIC Fall Rochester Planning Workshop, University of Rochester, NY, October 27–29: Shapiro, Kumar.

Second Annual RADC Technology Fair, Utica, New York, November 15, 1988: Shapiro.

2.12 NLP PUBLICATIONS IN 1988

Shapiro, S. C. and Rapaport, W. J., "Models and Minds: A Reply to Barnden", Northeast Artificial Intelligence Consortium Technical Report TR-8737, Department of Computer Science, SUNY at Buffalo, 13pp.

Kumar, D., Ali, S. and Shapiro, S. C., "Discussing, Using, and Recognizing Plans in SNePS Preliminary Report - SNaCTor: An Acting System" in *UBGCCS-88 Proceedings of the Third Annual UB Graduate Conference on Computer Science* Technical Report 88-03, Department of Computer Science, SUNY at Buffalo, 62-69.

Kumar, D., Ali, S. and Shapiro, S. C., "Discussing, Using, and Recognizing Plans in SNePS-Preliminary Report", *Modern Trends in Information Technology-Proceedings of the Seventh Biannual Convention of South East Asia Regional Computer Confederation,* (SEARCC 88), 177-182.

Sibun, P., "Directing the Generation of Living Space Descriptions", *COLING-88, The international Conference on Computaitonal Linguistics*, Budapest, Hungary.

Shapiro, S. C. "Representing Plans and Acts", Proceedings of the Third Annual Workshop on Conceptual Graphs, American Association for Artificial Intelligence, Menlo Park, CA, 3.2.7-1 - 3.2.7-6 and appendix.

2.13 CONFERENCES PARTIALLY SUPPORTED BY RADC IN 1988

Third Annual University Of Buffalo Graduate-Conference in Computer Science (UBGCCS-88)

Lynda Spahr was on the organizing committee for the Third Annual University Of Buffalo Graduate-Conference in Computer Science (UBGCCS-88), held in the the UB Center for Tomorrow on March 15, 1988. Six speakers from SUNYAB and two each from the Universities of Rochester, Waterloo, and Toronto participated. Scott Campbell, with Paul Palumbo, edited Tech Report 88-03, "UBGCCS-88 Proceedings of the Third Annual UB Graduate-Conference of Computer Science", S.S. Campbell, P.W. Palumbo, eds.

Deepak Kumar presented at the conference the paper, "Discussing, Using and Recognizing Plans in SNePS Preliminary Report - SNACTor: An Acting System", by D. Kumar, S. Ali and S. C. Shapiro.

Twenty-sixth Annual Meeting of the Association for Computational Linguistics (ACL-88)

The Twenty-sixth Annual Meeting of the Association for Computational Linguistics was held on the SUNY at Buffalo North Campus, June 7-10, 1988, partially supported by RADC through the NAIC. The local arrangements were co-chaired by Dr. William J. Rapaport and by Lynda Spahr, secretary at UB for the Northeast Artificial Intelligence Consortium. Other RADC sponsored volunteers included Jiah-shing Chen, Scott S. Campbell, David Satnik, Deepak Kumar and Syed Ali.

Founded in 1962, the Association for Computational Linguistics is the primary scientific and professional society for natural-language processing research and applications. A European chapter was established in 1982.

Approximately 330 linguists and artificial-intelligence researchers from universities and industry in the U.S., Canada, and overseas attended this conference, which offered an unparalleled opportunity for our faculty and graduate students.

Some of the topics that were discussed included: analogy, computerized lexicography, discourse and narrative, machine translation, mathematical linguistics, natural-language interfaces to databases, natural-language generation, parsing, speech-act theory, speech perception and production, syntax and semantics of natural languages, tense and aspect, and text processing.

There were also demonstrations of natural-language understanding systems by researchers from CUBRC, Cornell University, IBM, NYU, UB's SNePS Research Group, SRI International, Sun Microsystems, Unisys, and USC/ISI. The arrangements for these were made by Scott Campbell.

This was also the most successful ACL conference in terms of external support, with IBM, DEC, Barrister, Calspan, and CUBRC donating a total of $4700 towards the conference.

Other financial support came from the SUNY Buffalo Conferences in the Disciplines, FNSM, FSS, the Graduate Group in Cognitive Science, and the GRI in Cognitive and Linguistic Sciences.

APPENDIX 2.A

SNePS-2 USER'S MANUAL

2.A SNePS-2 USER'S MANUAL

2.A.1 Introduction

General

SNePS (the Semantic Network Processing System) is a system for building, using, and retrieving from propositional semantic networks. SNePS-2, described in this manual, has been implemented in Common Lisp, and runs on Texas Instruments Explorer Lisp Machines, Symbolics Lisp Machines, HP9000 AI Workstations, VAX 11/785's, a Sperry 7000/40, and an Encore Multimax. SNePS-2 differs in several respects from its predecessor, now called SNePS-79, mostly because of theoretical decisions that were made since SNePS-79 was implemented.

A semantic network, roughly speaking, is a labeled directed graph in which nodes represent concepts, arc labels represent binary relations, and an arc labeled R going from node n to node m represents that the concept represented by n bears the relation represented by R to the concept represented by m.

SNePS is called a *propositional* semantic network because every proposition represented in the network is represented by a node, not by an arc. Relations represented by arcs are called *non-conceptual relations* and may be thought of as part of the syntactic structure of the node they emanate from. Whenever information is added to the network, it is added in the form of a node with arcs emanating from it to other nodes.

Each concept represented in the network is represented by a unique node. This is enforced by SNePS-2 in that whenever the user specifies a node to be added to the network that would look exactly like one already there, in the sense of having the same set of arcs going from it to the same set of other nodes, SNePS-2 retrieves the old one instead of building the new one.

The core of SNePS-2 is a system for building nodes in the network, retrieving nodes that have a certain pattern of connectivity to other nodes, and performing certain housekeeping tasks, such as dumping a network to a file or loading a network from a file.

SNIP, the SNePS Inference Package, interprets certain nodes as representing reasoning rules, called *deduction rules*. SNIP supports a variety of specially designed propositional connectives and quantifiers, and performs a kind of combined forward/backward inference called *bi-directional* inference.

SNaLPS, the SNePS Natural Language Processing System, consists of a morphological analyzer, a morphological synthesizer, and a Generalized Augmented Transition Network (GATN) Grammar interpreter/compiler. Using these facilities, one can write natural language (and other) interfaces for SNePS.

The command language described in this manual is called SNePSUL, the SNePS User Language. It is a Lispish language, usually entered by the user at the top-level SNePSUL read-eval-print loop, but it can also be called from Lisp code or from GATN arcs. This manual follows the style of Guy Steele's COMMON LISP book, and assumes that the reader

is familiar that book and with Common Lisp.

Commands and Environments

A SNePSUL *command* is classified according to its role either as a *procedure* or as a *function*. A *procedure* is a command that performs some action but returns nothing, using the Common Lisp (**values**) function. A *function* is a command that always returns some value, possibly after having performed some action as a side effect. A function is implemented directly as a Lisp function. For every SNePSUL command *c*, whether procedure or function, (**get** *c* **=command**) has the value **t**.

A command is also classified according to the environment(s) in which it may legally appear. A procedure can be entered only at the top level of SNePSUL. A function, however, may appear in many different environments. For each environment, there is a symbol which appears on the property list of commands that are legal in that environment. The five environments and their symbols are:

The top level of SNePS-2	**=topcommand**
A *relation-set* position embedded in a command	**=rsfunction**
A *node-set* position in **build**	**=bnsfunction**
A *node-set* position in **find** or **findassert**	**=fnsfunction**
A *node-set* position in any of the other commands	**=onsfunction**

If *c* is a command and *s* is the symbol of an environment, (**get** *c s*) is **t** if *c* is legal in *s*, and is **nil** if *c* is not legal in *s*.

Finally, a command can be classified according to the relation between its position and the position of its arguments in the input line.

Most commands have an arbitrary number of arguments. They are called *prefix commands*, because they can only be entered using Cambridge prefix notation:

(*prefix-command argument ... argument*).

Some two-argument commands can be entered in infix position, and so are called *infix commands*. When an infix command is used in infix position, SNePS rearranges the input line to transform the form into a prefix form. Precedence is always from left to right. An infix command can be used as

(*infix-command argument argument*)

or as

argument infix-command argument

with no parentheses.

Since SNePS always remembers the result of the last top-level function, an infix command can also be used as

infix-command argument

in which case SNePS recalls the result of the last function and makes it the first argument for the infix command before rearranging the form to the prefix notation.

Similarly, some one-argument commands can be entered in postfix position and therefore are called *postfix commands.* A postfix command can be used as

(*postfix-command argument*)

or as

argument postfix-command

with no parentheses, or just as

postfix-command

in which case the result of the last function is used as argument.

Another kind of one-argument command, called *macro commands*, have one-character names and are used as

macro-command argument

with no parentheses, and preferably with no space between the command and the argument. Before passing it to the evaluator, the SNePS reader expands this form to a standard Cambridge prefix form.

Types of Nodes

There are four types of nodes in the SNePS network: base, variable, molecular, and pattern.

Base nodes are distinguished by having no arcs emanating from them. A base node may be created by the user's referring to it by name in the proper context. In such a case, the name of a base node can be any Lisp symbol. If a number is used, the node's name is a symbol whose symbol-name is a string of the characters that makes up the number. If a string is used, the node's name is the symbol whose symbol-name is that string. A base node may also be created using the # macro command, in which case the node's name is Bx, where x is some integer. A base node is assumed to represent some conceptual individual, object, class, property, etc. It is assumed that no two base nodes represent the same conceptual individual.

Variable nodes also have no arcs emanating from them, but represent arbitrary individuals or propositions, in much the same way that logical variables do. Variable nodes are created using the $ macro command. The name of a variable node is Vx, where x is some number.

Molecular nodes and pattern nodes have arcs emanating from them. Molecular nodes may represent propositions, including rules, or "structured individuals." A molecular node that represents a proposition may be *asserted* or *unasserted.* Pattern nodes represent arbitrary propositions or arbitrary structured individuals, and are similar to open sentences in predicate logic. Pattern nodes and unasserted molecular nodes are created by the `build`

function. Asserted molecular nodes are created by the **assert** function. An unasserted molecular node may be asserted by using the ! postfix command. The name of a pattern node is Px, where x is a number. The name of a molecular node is Mx, where x is a number. The name of an asserted molecular node is printed with a suffix of !.

Once any node is created, it may be referred to by its name. It is not necessary to include the ! suffix to refer to an asserted molecular node.[1]

SNePSUL Variables

SNePSUL, the SNePS User Language, has variables which are entirely distinct from SNePS variable nodes. The value of a SNePSUL variable is always a set of objects, **nil** if nothing else. A SNePSUL variable may be given a value with the **?**, **#**, or **$** macro commands, or with the = infix command. The value of a SNePSUL variable is obtained by using the * macro command. SNePSUL variables created and maintained by SNePS are:

assertions	The set of asserted nodes in the network.
commands	The set of SNePSUL commands.
nodes	The set of all nodes in the network.
relations	The set of defined arc labels.
variables	The set of SNePSUL variables.
varnodes	The set of variable nodes in the network.

2.A.2 SNePSUL Commands

Entering the SNePS Environment

To enter the SNePS environment on the UNIX time-sharing mini-computers named Marvin, Sybil, and Gort at SUNY/Buffalo, type the shell command **~snerg/bin/sneps2**; on the TI Explorers in the Department of Computer Science at UB, just evaluate **(sneps)**.

Entering and Leaving SNePS

The commands in this section move the user between the SNePSUL evaluator and the Common Lisp evaluator. Although every SNePSUL function is a Common Lisp function, the SNePSUL loop provides certain special facilities, so it is best to be in the proper top-level loop for extended work.

(sneps)
Lisp function that brings the user into the SNePS read-eval-print loop.

(lisp)
SNePSUL function that returns the user to the LISP evaluator.

[1] Currently, including the ! suffix in some contexts breaks SNePS, so you are advised not to think of it as part of the name for input purposes

-

SNePSUL command that causes the next form to be evaluated by Lisp.

--

SNePSUL command that puts the user into an embedded Lisp read-eval-print loop until the next occurrence of the form --, whereupon the user is returned to the SNePSUL loop.

(exit)
SNePSUL function that terminates Common Lisp and returns the user to the operating system. **Warning:** This command is not currently bound to any function.

Using Auxiliary Files

The commands in this section provide for the use of auxiliary files for the storage of networks or of sequences of commands.

(outnet *file***)**
Stores the current network on the *file* in a special SNePS format. The syntax for the file specification is machine dependent.

(innet *file***)**
If *file* was created by a call to **outnet**, the current network will be initialized to the one stored on *file*. **Note:** innet rewrites the entire network and several SNePSUL variables, so it cannot be used to combine several networks. An error message is issued if *file* is not in the appropriate format.

(intext *file***)**
Reads a sequence of SNePSUL commands from the *file* and executes them, without echoing them.

(demo *file* **&key :pause)**
Reads a sequence of SNePSUL commands from the *file*, echoes them, and executes them. If **:pause** is **t**, SNePS will pause before each command is executed until the user enters a carriage return. Each command may be preceded by an arbitrary number of comment lines each beginning with ";". These lines will be echoed. **demo** prints the SNePS timing of each command separately, as well as the timing of the entire demonstration. **Warning:** Currently, the input *file* must end on a fresh line, not on the same line as the last command.

Defining Arc Labels

By *relation* in this manual, we mean any non-conceptual relation used to label network arcs. Therefore "relation" and "arc label" are used interchangeably. Whenever an arc

labelled R goes from node x to node y, SNePS considers an arc labelled $R-$ to go from y to x. Relation names ending in the character #\- are reserved for this "reverse arc" or "converse relation" labelling. Therefore no relation name may end with a #\-. The term *relation* always refers to a normal, "forward" arc label. We will use the term *unitpath* to mean either a relation name or the name of its converse relation.

(define {*relation*}*)
Defines each *relation* to be an arc label. The name of a relation must not end in the character #\-. Each *relation* is added to the SNePSUL variable **relations**. An informative message is given if a relation has previously been defined. Initially, SNePS has a set of relations defined as if the following had been executed:

```
(define  forall exists pevb
         min max thresh threshmax emin emax etot
         ant &ant cq dcq arg default)
```

For uses of the predefined relations, see Sections 3.1.1, "Connectives," and 3.1.2, "Quantifiers."

(undefine {*relation*}*)
Undefines each *relation*. If any *relation* is being used in the current network, the arcs are not removed from the network structure, but they do become undefined. **undefine** is most useful in correcting typographical errors in calls to define.

(define-path {*relation path*}*)
Declares the path-based inference rule,

$$\forall(n_1, n_2) path(n_1, n_2) \Rightarrow relation(n_1, n_2).$$

I.e., if a path of arcs specified by *path* is in the network going from node n_1 to node n_2, then the single arc labelled by *relation* is inferred as going from node n_1 to node n_2. See the following section for the syntax of *path*. No *relation* may have more than one path-based inference rule for it at any time. This is not a restriction, since a disjunction of paths is also a path. **Warning:** A path-based inference rule will not be expanded recursively. *I.e.*, no relation (or converse relation) in the path will be expanded even if a path-based inference rule has been declared for it. **Warning:** Currently, if *relation* already has a path-based inference rule for it, you will not be able to change it.

(undefine-path {*relation path*}*)
Deletes the given path-based inference rules. **Warning:** Currently, this is not implemented.

Syntax and Semantics of Paths

A *unitpath* is simply a single arc followed in the forward or the reverse direction. A *path* can be a sequence of unitpaths, or a more complicated way of getting from one node to

another. Keep in mind the distinctions between *relation*, *unitpath*, and *path*, since there are places where it matters.

unitpath ::= *relation*
Any single arc *relation* is also a *unitpath*.

unitpath ::= *relation*-
If R is a relation from node x to node y, then R- is a unitpath from y to x.

path ::= *unitpath*
Any single arc, either forward or backward, is a *path*.

path ::= (**converse** *path*)
If P is a path from node x to node y, then (**converse** P) is a path from y to x.

path ::= (**compose** *path**)
If $x_1, \ldots, x_n$ are nodes and P_i is a path from x_i to x_{i+1}, then

(**compose** P_1 ... P_{n-1})

is a path from x_1 to x_n. **Note:** If the symbol ! appears between P_{i-1} and P_i, then x_i must be asserted. **Example:** After doing

(**build member socrates class man**),

the path

(**compose member- class**)

goes from **socrates** to **man**, but the path

(**compose member- ! class**)

doesn't. However, after doing

(**assert member socrates class man**),

both paths exist.

path ::= (**kstar** *path*)
If path P composed with itself zero or more times is a path from node x to node y, then (**kstar** P) is a path from x to y.

path ::= (**kplus** *path*)
If path P composed with itself one or more times is a path from node x to node y, then (**kplus** P) is a path from x to y.

path ::= (or {*path*}*)
If P_1 is a path from node x to node y or P_2 is a path from x to y or ...or P_n is a path from x to y, then (or P_1 P_2 ...P_n) is a path from x to y.

path ::= (and {*path*}*)
If P_1 is a path from node x to node y and P_2 is a path from x to y and ...and P_n is a path from x to y, then (and P_1 P_2 ...P_n) is a path from x to y.

path ::= (not *path*)
If there is no path P from node x to node y, then (not P) is a path from x to y.

path ::= (relative-complement *path path*)
If P is a path from node x to node y and there is no path Q from x to y, then

(relative-complement P Q)

is a path from x to y.

path ::= (irreflexive-restrict *path*)
If P is a path from node x to node y, and $x \neq y$, then

(irreflexive-restrict P)

is a path from x to y.

path ::= (exception *path path*)
If P is a path from node x to node y and there is no path Q from x to y with length less than or equal to the length of P, then (exception P Q) is a path from x to y.

path ::= (domain-restrict (*path node*) *path*)
If P is a path from node x to node y and Q is a path from x to node z, then

(domain-restrict (Q z) P)

is a path from x to y.

path ::= (range-restrict *path* (*path node*))
If P is a path from node x to node y and Q is a path from y to node z, then

(range-restrict P (Q z))

is a path from x to y.

Building Networks

The commands of this section add information to the network, either in the form of a node, a node and some arcs, or an assertion tag. It is not possible to add just an arc to the network. Isolated nodes cannot be added to the network, so the commands **#** and **$** can only be used within the lexical context of a **build**, **assert**, or **add**.

We will use the term *wire* to mean a labelled arc and the node it points to. So a molecular node has a set of wires coming out of it.

(**build** {*relation nodeset*}*)
(**assert** {*relation nodeset*}*)
(**add** {*relation nodeset*}*)
Puts a node in the network with an arc labelled *relation* to each node in the following *nodeset*, and returns a singleton set containing the built node. The new node is added to the value of the SNePSUL variable **nodes**. If this new node would look exactly like an already existing node, *i.e.,* would have exactly the same set of wires emanating from it, then no node is built, but a singleton set containing the extant node is returned. **build** creates an unasserted node, unless an asserted node exists in the network with a superset of the wires of the new node, in which case the new node is also asserted. **assert** is just like **build**, but creates the node as an asserted node. **add** is just like **assert**, but, in addition, triggers forward inference. **Note:** where *relation* is specified in the syntax, neither a converse relation nor a non-unit path is allowed.

(**!** *node*)
A postfix command that asserts *node*, and returns a singleton set containing *node*.
(**assert** ...) is equivalent to (**!** (**build** ...)) and to (**build** ...)**!**.

#*symbol*
A macro command that creates a new base node, assigns a singleton set containing the new node as the value of the SNePSUL variable *symbol*, and returns that set. This may not be used at the top-level SNePSUL loop, since that would create an isolated node.

$*symbol*
A macro command that creates a new variable node, assigns a singleton set containing the new node as the value of the SNePSUL variable *symbol*, and returns that set. This may not be used at the top-level SNePSUL loop, since that would create an isolated node.

Deleting Information

The commands of this section delete information from the network, and are mainly intended for use after mistakes or when debugging.

(**erase** {*nodeset*}*)

Removes all nodes in all *nodesets* from the network along with any nodes that become isolated in the process (that is, all nodes which no longer have any arcs connected to them). Asks the user for permission to delete unasserted nodes dominated by nodes it erases. Refuses to delete nodes that have arcs coming into them.

(resetnet)
Reinitializes the network to the state in which no nodes have been built. Leaves relations defined and path-based inference rules declared.

(clear-infer)
Deletes any information placed in the "active connection graph" version of the network by SNIP. *I.e.*, all deduction rules are returned to their unactivated state as if no inference had yet been performed.

Functions Returning Sets of Nodes or of Unitpaths

The functions described in this section neither add to nor delete from the network. Rather, they compute and return sets either of nodes or of unitpaths.

({*node*}*)
A list of nodes at the top level of the SNePSUL loop, or in a context where a node set is required, is treated as an expression whose value is a set of the nodes in the list.

******symbol*
A macro command function which returns the set of nodes in the value of the SNePSUL variable *symbol.*

(^ *S-expression*)
The set of nodes obtained by evaluating the LISP *S-expression.*

(& *nodeset nodeset*)
Infix function that returns the intersection of the two *nodesets.*

(+ *nodeset nodeset*)
Infix function that returns the union of the two *nodesets.*

(- *nodeset nodeset*)
Infix function that returns the set of nodes in the first *nodeset* but not in the second *nodeset.*

(= *nodeset symbol*)
Infix function that assigns the *nodeset* to be the value of the SNePSUL variable *symbol.*

(> *unitpathset symbol*)
Infix function that assigns the *unitpathset* to be the value of the SNePSUL variable *symbol*.

(_ *nodeset unitpathset*)
Infix function that returns the set of those nodes in the *nodeset* which do not have any of the unitpaths in the *unitpathset* emanating from them.

Displaying the Network

The commands in this section are various ways of printing, or otherwise displaying, the information in the network.

(dump {*nodeset*}*)
Prints the name of each node in the *nodeset*, along with all arcs going from it or into it, and the nodes that each arc points to or from. For a complete dump of the network, execute (dump *nodes).

(describe {*nodeset*}*)
Similar to dump, but: describes only the molecular and pattern nodes in the *nodesets*; describes all molecular and pattern nodes dominated by nodes it describes; describes any node at most once—the second and later times, only the node's name is printed.

(surface {*nodeset*}*)
Generates a description of each node in each *nodeset* using the currently loaded GATN grammar starting in state g.

(ginseng)
(gi-dump {*nodeset*}*)
(gi-desc {*nodeset*}*)
Runs the Ginseng facility for drawing a graphical display of the SNePS network. First, the user must position and size the Ginseng Window using the mouse. Once that is done, two Ginseng menus are available: one, by clicking any mouse button on the Window background; the other, by clicking any mouse button on a node displayed in the Window. It is possible to go back and forth between the window running SNePS, and the Ginseng window, so the network can be modified in the SNePS window, and the modification may then be drawn in the Ginseng Window. ginseng gives an initially empty Ginseng window. gi-dump initializes the window to contain the specified nodes and all nodes they are connected to by a single arc. gi-desc initializes the window to contain the specified nodes and all nodes they dominate. **Warning:** Ginseng is currently available only on Texas Instruments Explorers.

Retrieving Information

The functions in this section find nodes in the network, and return them.

(**find** {*path nodeset*}*)
(**findassert** {*path nodeset*}*)
(**findconstant** {*path nodeset*}*)
(**findbase** {*path nodeset*}*)
(**findvariable** {*path nodeset*}*)
(**findpattern** {*path nodeset*}*)
Returns the set of nodes such that each node in the set has every specified *path* going from it to at least one node in the accompanying *nodeset*. (**find class (man greek)**) will find nodes with a **class** arc to either **man** or **greek**, whereas (**find class man class greek**) will find nodes with **class** arcs to both **man** and **greek**. **find** returns all appropriate nodes in the network; **findassert** returns only asserted nodes; **findconstant** returns only base or molecular nodes; **findbase** returns only base nodes; **findvariable** returns only variable nodes; **findpattern** returns only pattern nodes.

?symbol
May be used in any **find** function in place of a *nodeset*, to stand for "any node." The scope of these symbols is the outermost **find** function and all embedded **find** functions. After return of the outermost **find** function, *symbol* will be a SNePSUL variable whose value will be the set of nodes it matched.

(**deduce** [*numb*] {*relation nodeset*}*)
Like **findassert**, but uses SNIP to back-chain on any deduction rules in the network, and returns all inferred nodes that satisfy the specification. Note that only *relations* may appear in the specification, not any other unitpaths or paths. Neither may *?symbol* variables appear in the specification. The *numb* argument is optional. If *numb* is omitted, then deduce continues until no more answers can be derived. If *numb* is a single integer, it specifies the total number of answers requested. If *numb* is zero, no inference is done—only answers already in the network are returned. Otherwise, *numb* must be a list of two numbers, (*npos nneg*), and deduction terminates after at least *npos* positive and *nneg* negative instances are derived.

2.A.3 SNIP: The SNePS Inference Package

Automatic inference may be triggered using the function **deduce** (see Sec. 2.2.10), a generalization of **find**, or the function **add** (see Sec. 2.2.6), a generalization of **assert**. In order for these to accomplish anything, deduction rules must exist in the network. A deduction rule is a network structure dominated by a rule node. A rule node represents a logical formula of molecular nodes, using connectives and quantifiers.

Representing Rules

Rules are placed in the network with the **assert** and **add** commands. The arcs needed to build rules are predefined by SNePS.

Connectives

Connectives are the means by which simple propositions are compounded to make more complicated ones. In classical logic, this compounding is accomplished by use of standard connectives such as & (AND) and ∨ (OR). A number of disadvantages exist in using standard connectives in SNePS, primarily because of their binary nature and the size of the network needed to store representations with standard connectives. To avoid these problems, SNePS uses non-standard connectives. These non-standard connectives are as adequate as standard connectives, but they take arbitrarily large sets of arguments and express common modes of human reason simply. The non-standard connectives are: and-entailment, or-entailment, numerical entailment, andor, thresh, non-derivable, and default. SNePS determines which connective is being used from the set of predefined arcs emanating from the rule node, and, in the cases of andor, thresh, and numerical entailment, which have numerical parameters, by special arcs from the rule node to the parameters. An explanation of each connective follows.

And-Entailment $\{A_1,\ldots,A_n\}\ \&\Rightarrow\ \{C_1,\ldots,C_m\}$ means that the conjunction of the antecedents implies the conjunction of the consequents. An and-entailment rule is built with the SNePSUL command:

(assert &ant $(A_1,\ldots,A_n)$
cq $(C_1,\ldots,C_m)$)

Or-Entailment $\{A_1,\ldots,A_n\}\ \vee\Rightarrow\ \{C_1,\ldots,C_m\}$ means that the disjunction of the antecedents implies the conjunction of the consequents. An or-entailment rule is built with the SNePSUL command:

(assert ant $(A_1,\ldots,A_n)$
cq $(C_1,\ldots,C_m)$)

Note: or-entailment is more efficient than and-entailment, so if there is only one antecedent, use **ant** rather than **&ant**.

Numerical Entailment $\{A_1,\ldots,A_n\}i\Rightarrow\ \{C_1,\ldots,C_m\}$ means that the conjunction of any i of the antecedents implies the conjunction of the consequents. In other words, if i or more of the antecedents are true, then all of the consequents are true. A numerical-entailment rule is built with the SNePSUL command:

(assert thresh i
&ant $(A_1,\ldots,A_n)$
cq $(C_1,\ldots,C_m)$)

AndOr $\mathcal{N}_i^j\{P_1, \ldots, P_n\}$ means that at least i and at most j of the n propositions are true. An andor rule is built with the SNePSUL command:

(assert min i max j
arg $(P_1, \ldots, P_n)$)

The following special cases of andor are representations of standard connectives: $i = j = n$ is AND; $i = j = 0$ is a generalization of NOR; and $i = j = 1$ is a generalization of EXCLUSIVE OR.

Thresh $\Theta_i^j\{P_1, \ldots, P_n\}$ means that either fewer than i or more than j of the n propositions are true. j may be omitted, in which case it defaults to $n - 1$. A thresh rule is built with the SNePSUL command:

(assert thresh i threshmax j
arg $(P_1, \ldots, P_n)$)

If $i = 1$ and j is omitted, the thresh is a generalization of equivalence.

Non-derivable $\nvdash P$ means that P is not derivable in the current network. **Warning:** non-derivable has not yet been implemented in SNePS-2.

Quantifiers

Quantifiers permit the use of variables in deduction rules. The relations `forall` and `exists`, are predefined quantifier relations. They are used to point to variable nodes, indicating for which values of the variable node the rule holds. `forall` and `exists` represent universal and existential quantifiers, respectively. SNePS-2 uses restricted quantification, which means that every quantified expression must have a restriction as well as a scope.

The Universal Quantifier

$$\forall(x_1, \ldots, x_n)\{R_1(x_1), \ldots, R_n(x_n)\} : \{P_1(x_1, \ldots, x_n), \ldots, P_m(x_1, \ldots, x_n)\}$$

means that for every substitution, $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ for which the following conditions hold

- t_i satisfies the restriction $R_i, 1 \leq i \leq n$
- $t_i \neq t_j$ whenever $i \neq j$
- t_i does not occur in the original rule, $1 \leq i \leq n$

$P_i(x_1, \ldots, x_n)\sigma, 1 \leq i \leq m$ is true. There may be fewer restrictions than variables if some restriction contains more than one variable free, as long as every variable occurs in at least one restriction. A universally quantified rule is built with the SNePSUL command:

(assert forall $(x_1, \ldots, x_n)$
&ant $(R_1(x_1), \ldots, R_n(x_n))$
cq $(P_1(x_1, \ldots, x_n), \ldots, P_m(x_1, \ldots, x_n)))$

The first occurrence of a variable must be preceded by the **$** macro, and subsequent occurrences must be preceded by the * macro.

If there is only one restriction, **ant** should be used instead of **&ant**.

The Existential Quantifier

$$\exists(x_1, \ldots, x_n)\{R_1(x_1), \ldots, R_n(x_n)\} : \{P_1(x_1, \ldots, x_n), \ldots, P_m(x_1, \ldots, x_n)\}$$

means that for some substitution, $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ for which the following conditions hold

- t_i satisfies the restriction $R_i, 1 \leq i \leq n$
- $t_i \neq t_j$ whenever $i \neq j$
- t_i does not occur in the original rule, $1 \leq i \leq n$

$P_i(x_1, \ldots, x_n)\sigma, 1 \leq i \leq m$ is true. There may be fewer restrictions than variables if some restriction contains more than one variable free, as long as every variable occurs in at least one restriction. **Warning:** the existential quantifier has not yet been implemented in SNePS-2.

The Numerical Quantifier

$$_k\exists_i^j(x_1, \ldots, x_n)\{R_1(x_1), \ldots, R_n(x_n)\} : \{P_1(x_1, \ldots, x_n), \ldots, P_m(x_1, \ldots, x_n)\}$$

means that of the k substitutions, $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ for which the following conditions hold

- t_i satisfies the restriction $R_i, 1 \leq i \leq n$
- $t_i \neq t_j$ whenever $i \neq j$
- t_i does not occur in the original rule, $1 \leq i \leq n$

between i and j of them also make $P_i(x_1, \ldots, x_n)\sigma, 1 \leq i \leq m$ true. There may be fewer restrictions than variables if some restriction contains more than one variable free, as long as every variable occurs in at least one restriction. **Warning:** the numerical quantifier has not yet been implemented in SNePS-2.

The Uniqueness Principle for Variables Currently, the Uniqueness Principle is not enforced by SNePS for variables. Therefore, it is advised that the Uniqueness Principle for variables be followed by the SNePSUL user as a matter of style. This should be done as follows. Every restriction used in a restricted quantifier should have a series of variables, $x_1^R, x_2^R, \ldots$ Every rule that uses R once should use x_1^R as its variable. A rule that uses the restriction R more than once should use x_1^R in the first use of R, x_2^R in the second use of R, etc. This can be done by using the $ macro to create each variable node the first time the restriction occurs, and the * macro on all subsequent occasions, including subsequent rules. For example, the two rules "Every dog is a pet" and "Every dog hates every cat" might be entered as follows, assuming that the restrictions $Dog(x)$ and $Cat(y)$ have not previously been used in the network:

```
(assert forall $dog1
        ant   (build member *dog1 class dog)
        cq    (build member *dog1 class pet))
(assert forall (*dog1 $cat1)
        &ant ((build member *dog1 class dog)
              (build member *cat1 class cat))
        cq    (build agent *dog1 act hates object *cat1))
```

APPENDIX 2.B

REFERENCE TO SITUATIONS

Reference to Situations

Penelope Sibun
Scott D. Anderson
David Forster
Beverly Woolf

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Email: sibun@cs.umass.edu *or* penni@umass.bitnet

Abstract

We discuss the definition and processing of words such as "situation," "example," and "case." In this paper, we restrict the discussion to the word "situation." We have concluded that a *Situation*, the referent of this word, is an aggregate of model objects picked out by a *Situation Index*. Situations are states of affairs characterized by concreteness and tension. We show further that context must inform the comprehension of this difficult lexical item; and we conclude with a discussion of how reference to situations interacts with text structure.

We thank James Pustejovsky for his many helpful suggestions and criticisms.

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 under contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

1 Introduction

Words such as "situation" are important because they occur frequently in natural discourse, and they are interesting because they are like anaphoric terms ("he," "it") and deictic terms ("this," "that"), but they have a much richer semantics. Consider the following usage:

> Penni and Scott are writing an IJCAI paper. Their first draft was shot down by their fellow researchers. In this situation, they squabbled more than usual.

Most people would agree that this paragraph contains a description of a situation, and more importantly, that the lexical item "situation" successfully refers to it. However, understanding the exact reference of the word "situation" is complicated by three factors:

1. "Situation" places significant, but ill-defined, constraints on its possible referents. This example shows elements of concreteness (particular people writing a particular paper), states (a bad first draft), and change (such as increased squabbling). These are much more complex constraints than the simple number, person, and gender features of pronouns.

2. "Situation" is nevertheless a weak descriptor, because situations come in so many different kinds—different sets of entities can be referred to as a situation, depending on the context.

3. Situations might not exist *a priori*, but instead be created *a posteriori*. That is, the set of objects that make up the situation may not previously have been aggregated. The objects aggregated by "situation" reside in the Model,[1] which is constructed in the process of understanding the text.

This paper attempts to define the semantics and processing of situations.

[1]This Model is informed by, but is distinct from, our domain models, which contain world knowledge of various possible domains of discourse. The Model may contain explicit structural or lexical information; this needs to be saved for some understanding tasks (for example, Webber's discourse anaphora (1987)), but it is not necessary for processing situations.

2 Types of Reference

Model Anaphora, which we define to be the determination of the reference of words such as "situation," naturally extends other work on anaphora. Many researchers have studied pronoun anaphora (coreference between a noun phrase and a personal pronoun): Hobbs (1978), Grosz and Sidner (1986), Hankamer and Sag (1976), and Webber (1983). Work on temporal anaphora (coreference between temporal expressions) has been done by Partee (1984) and Hinrichs (1986), among others. Webber has also worked on the reference of demonstratives such as "this" and "that"; she has termed this variously as "event reference" (1987) and "discourse anaphora" (1988). The 1987 paper also introduced the notion of *individuating reference*, which plays a key role in our Model Anaphora.

Text comprehension involves *reference*, which we define as the link from an expression in the text (for example, a noun phrase or a proposition) to a Model Object,[2] called the *referent*. The arrows in Figure 1 are examples of such reference links.

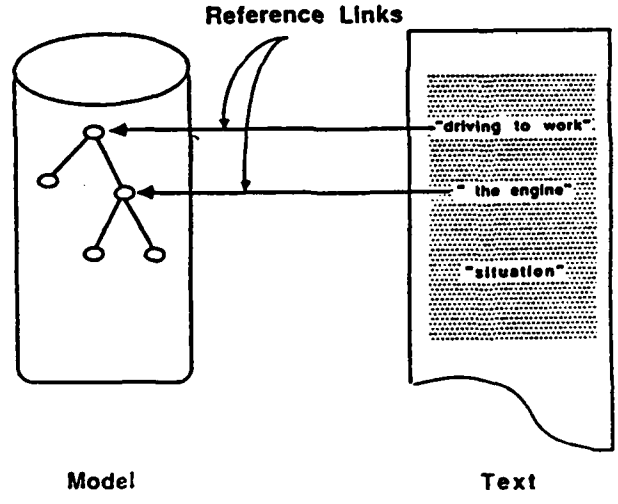

Figure 1: The reference links from the text to the Model.

Sometimes, the referent of an expression is a *set* of things in the Model. Such a set acts as an *individual*, in that it can be referred to, can have qualities attributed to it, and can participate in relations. Webber (1987)

[2]A Model Object is any first-class object in the Model (essentially, a first-class object is anything that can be pointed to). Therefore, Model Objects include relations, events, states, and so forth, as well as physical objects. Recall that the Model records our understanding of the text, and is distinct from domain models.

calls a reference that creates a new individual an *individuating reference*, depicted in Figure 2. One can also view this as creating a new first-class object, since only first-class objects can be referred to and predicated.

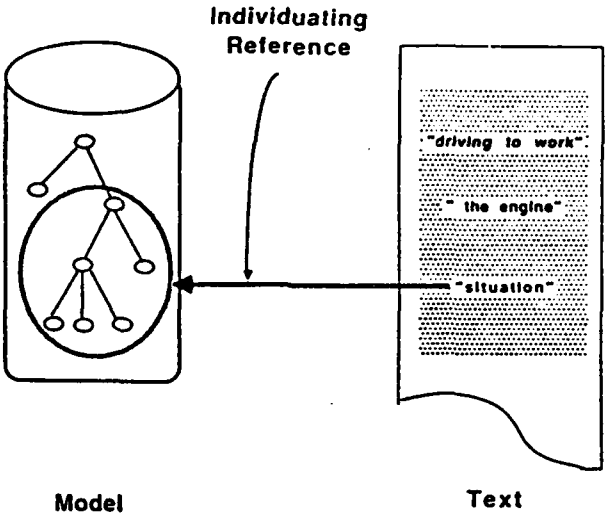

Figure 2: An expression referring to a set of objects as though it were an individual. The reference that creates this set is called an *individuating reference.*

Not all aggregate model objects are the result of individuating reference: some aggregates already exist as part of a domain model. For instance, the story of Romeo and Juliet is clearly an aggregate, being composed of many people and events, but it is easily available as a referent in discourse without being created anew.

We believe that a situation is represented as an aggregate model object and that the word "situation" is often used as an individuating reference. There are two issues to address: defining just what can be termed a situation, and processing a reference to a situation (which may involve an individuating reference). The next section deals with the problem of a proper definition of this term, and Section 4 deals with processing.

3 The Definition of Situations

We have identified several factors which influence whether an expression describes a situation. Consider examples (1) through (4) below. When asked to judge which examples can be characterized as situations, speakers generally agree that (4) can be a situation, but (1) cannot. However, there is less

agreement on whether the intermediate expressions may refer to situations.

(1) running
(2) running the Boston Marathon
(3a) James running the Boston Marathon
(3b) James having run the Boston Marathon
(4) James having run the Boston Marathon last April

It is possible to imagine contexts in which any of these phrases describes a situation, but in some cases, the contexts are more natural and more available than in others. We believe that these expressions differ by their *concreteness*.

Concreteness is characterized in the psychological literature by imageability (for example, Paivio *et al.*, 1968). In our examples, the increasing specificity supplies the details that aid in building up a mental image. Specificity, however, is not the right measure of a situation. Consider example (5):

(5a) playing a violin
(5b) playing a Stradivarius

Most people judge these as equally good (or bad) as situations, yet (5b) is clearly more specific, though it is no more concrete (imageable). This is similar to Rosch's (1976) notion of *basic level categories*; one formulation of a basic level is that it is the highest level for which the *prototype* is imageable (hence, concrete). Thus, concepts at or below the basic level are equally concrete, and therefore their concreteness equally affects judgements of whether something is a situation.

Intuitively, a situation is a state of affairs—a description of the world (or some small part of it) at some point in time. Note that the questions "What is the situation with X?" and "What is the state of X?" will elicit similar, if not identical, descriptions. Certainly, the description may combine present and past actions, especially those that are related (causally or otherwise) to the current state of affairs:

> The situation is that I dialed this long-distance number and it's been over thirty seconds and it still hasn't rung. Strange.

Clearly, this situation includes the past action of dialing the phone. On the other hand, a situation need not mention past actions. For example, the situation at the beginning of *Romeo and Juliet* can adequately be described as the feud between the Montagues and the Capulets, without mentioning any particular actions.

A situation usually contains *tension*, which we take as *potential for change*. In Example (6) below, (6a) is a better situation than (6b) because there is a greater potential for change. Most people consider (6b) too uninteresting to be a situation at all. Of course, in certain contexts, (6b) could be considered a situation: a textbook on erosion might discuss the changes in that situation.

(6a) A boulder teetering on the edge of a cliff.
(6b) A boulder sitting in the middle of a plain.

Example (6) also demonstrates that tension need not result from agents and their goals, as it does in the Romeo and Juliet situation.

We conclude that situations are states of affairs characterized by concreteness and tension. Concreteness reflects how easily people can construct a mental image of the situation. Tension reflects people's judgements of the potential for change.[3] Obviously, judgements of these characteristics will vary from person to person; nevertheless, they put significant constraints on what can be a situation.

4 Processing

Situations are commonly referenced with phrases such as "the situation with X" or "the X situation." Sometimes, depending on X, more specific phrases are used, as with "the situation in the Middle East," which is preferable to "the situation with the Middle East." These modifying phrases include the *Situation Index* for each of these Situations. A *Situation Index* is a key that picks the situation out of the Model, distinguishing foreground from background, so to speak.

The situation index is usually some common or unifying aspect of the situation. Thus, if we speak of "the IJCAI paper situation," (see our first example), the situation index unites the situation. This situation is also successfully referred to by "the situation with Penni." Indeed, the situation index need not be explicitly mentioned in the text. For example:

> I was driving to work with the carpool, as usual, and the engine threw a rod. It's shot, so now I've got to get a new one. That's the situation with my car.

[3]Some situations, unfortunately, seem to be ones in which people nevertheless judge there to be little potential for change. For example, consider the situation of being trapped in the center of a row at a boring lecture.

This reference is successful because the mention of "driving" and "engine" bring "car" into *focus* (see Sidner 1979). Furthermore, replacing the referring phrase with "the situation with the carpool" does not as easily refer to the car situation. Therefore, we conclude that elements in focus, regardless of explicit mention in the text, are what are available as situation indexes.

We take references that make explicit mention of the situation index to be the canonical case; we process other references by first determining the situation index from the context, thereby reducing them to the canonical case. Thus, Model Anaphora is a two-step process:

1. **Determine the situation index.** Where this is explicitly mentioned, the determination is trivial. Otherwise, the situation index must be computed from the context.

2. **Using the situation index, search the stack of focus spaces and look for matches.** Since we take the focus spaces to comprise Model Objects, the search will be through a subspace of the Model. The matching Model Objects will yield a set of candidates, some of which will be discarded because they are not states of affairs or because they lack concreteness or tension. If this does not reduce the set of candidates to a single situation, we choose the most recent candidate.

Our current research concerns the comprehension of the following paragraph, so we will use it to illustrate Model Anaphora:

> Nancy asked Tom if an inanimate object, such as a table, can exert a force. Tom said he didn't think so. Nancy pointed to a pile of books on the table and asked if the table exerts an upward force on the books. Tom said no. Nancy placed the books on Tom's hand. Tom had to exert a great force to keep the books steady. Nancy asked Tom to compare the two situations. Tom said the table must also have exerted a force on the books.

Nancy definitively sets the topic of the paragraph with her first question to Tom. This places "inanimate objects" and "force" in focus, as well as marking them as belonging to the topic. Because of our domain knowledge of physics, we know that her second question, which is more specific, is subsumed by her first, and does not change the topic. When Nancy refers to "the two situations" without supplying a situation index, either "inanimate objects" or "force" is probably the index she has in mind.

Supposing we take "force" as the situation index, we search in the Model for Model Objects that match it. We find the first situation because "push" is a kind of force and Nancy has asked Tom whether the books push up on the table. Hence, *the books on the table* is a situation. The second situation is found because force is explicitly mentioned, hence *the books on Tom's hand* is also a situation.[4] Although force is mentioned in the first sentence, "whether an inanimate object, such as a table, can exert a force," this is discarded as a situation because it lacks concreteness.

We have outlined the two phases of processing Model Anaphora and have shown how we can find the referents for "situation" in an sample text. The first step in the process is finding a situation index, which we have defined as the key that picks the situation out of the Model. The second step involves searching the focus stack for occurrences of the situation index and determining whether any of these is part of a situation.

5 Conclusion

Model Anaphora is based on reference to objects in a Model, as opposed to the structure of the text. For instance, we distinguish our work from Webber's research showing that demonstrative pronouns refer to discourse segments (1988). While many descriptions of situations will correspond to discourse segments, other descriptions will be discontinuous or overlapping. An example of the latter is the following, in which Frieda's situation overlaps with Bjorn's:

Helga loves Bjorn who loves Frieda who loves Bant.

Because of our reliance on focus spaces, the structure of the text will affect the availability of a situation to a bare reference (a referring phrase with no situation index). For example, text intervening between the description of a situation and the bare reference will usually make the reference confusing.

In conclusion, we claim that situations are states of affairs that exhibit concreteness and tension, and that the process of referring to them involves a situation index. We believe that Model Anaphora is a phenomenon common to a large class of nouns, including "case," "disaster," and "example." These nouns differ somewhat in their semantics, but they all make individuation references to sets of Model Objects.

[4] Both situations are larger than we have described them here. For instance, Nancy would include in her representations the forces involved in the two situations. Tom will, too, once he understands the concept.

6 References

Grosz, B. and C. Sidner, "Attention, Intentions, and the Structure of Discourse." In *Computational Linguistics*, Vol. 12, No 3, pp. 175-204, 1986.

Hankamer, J. and I. Sag. "Deep and Surface Anaphora." In *Linguistic Inquiry*, 7(3), pp. 391-426, 1976.

Hinrichs, E. "Temporal Anaphora in Discourses of English." *Linguistics and Philosophy*, 9(1), pp. 63-82, 1986.

Hobbs, J. "Resolving Pronoun References." In *Lingua* 44 pp. 311-338, 1978. Also in B. Grosz, K. Sparck-Jones, and B. Webber, eds., *Readings in Natural Language Processing*, pp. 339-352, Morgan Kaufman Publishers, Inc., 1986.

Paivio, A., J. Yuille, and S. Madigan. "Concreteness, Imagery, and Meaningfulness Values for 925 Nouns." In *Journal of Experimental Psychology Monograph Supplement*, 76, pp. 1-25, 1968.

Partee, B. "Nominal and Temporal Anaphora." In *Linguistics and Philosophy*, 7(3), pp. 243-286, 1984.

Rosch, E., C. Mervis, W. Gray, D. Johnson, and P. Boyes-Braem. "Basic Objects in Natural Categories." In *Cognitive Psychology*, 8, pp. 382-439, 1976.

Sidner, C. *Towards a Computational Theory of Definite Anaphora Comprehension In English Discourse.* Technical Report TR-537, Massachusetts Institute of Technology, 1979.

Webber, B. "So What Do We Talk About Now?" In M. Brady and R. Berwick, Eds., *Computational Models of Discourse*, The MIT Press, pp. 331-371, 1983.

Webber, B. "Event Reference." In *Position Papers for TINLAP-3: Theoretical Issues in Natural Language Processing-3*, Las Cruces, NM, pp. 137-142, 1987.

Webber, B. "Discourse Anaphora." In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, pp. 113-122, 1988.